

Projeto Final Ciência de Dados

March 12, 2021

1 Ciência de Dados - Trabalho final

1.1 Objetivo:

- Criação de um modelo de classificação para identificação de malwares das categorias *Adware*, *Ransomware*, *Scareware* e *SMS Malware* em aplicações Android.

1.2 *Dataset* utilizado:

- *Dataset* obtido da [UNB \(University of New Brunswick\)](#) chamado CIC-AndMal2017, contendo dados de fluxo de tráfego rede de 10.854 aplicações Android sendo 4.354 malware e 6.500 de aplicações benignas.
- Nos arquivos relacionados ao *dataset* há além dos CSVs com dados de fluxo de tráfego as APKs das aplicações.
- Mais detalhes sobre o *dataset* pode ser visto na seção “[Exploração de dados](#)” do projeto no github.

1.3 Bibliotecas implementadas

- Alguns códigos foram implementados para auxiliar o processamento dos dados e experimentos e são listados a seguir com seus respectivos links.
 - [manifest_script.py](#)
 - [funcoes_uteis.py](#)

2 Pré-processamento dos dados

- Os dados de fluxo de tráfego estão armazenados em arquivos CSVs separados em pastas por categoria e família.
- Todos os arquivos possuem a mesma estrutura com 87 colunas.
- Para fazer o treinamento dos modelos considerando o objetivo deste trabalho, faz-se necessário a junção desses arquivos em apenas um. Este processo é apresentado na sequência desse *notebook*.

```
[1]: # Importação de bibliotecas
import pandas as pd
import numpy as np

from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
```

```

from sklearn.neural_network import MLPClassifier

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

# Códigos implementados pelo autor
from lib import funcoes_uteis

```

```

[4]: path = '/home/efbaro/Documentos/dados/CSVs/'
# Obtém recursivamente o caminho para todos os arquivos CSVs
# a partir do ponto inicial passado como parâmetro
files = funcoes_uteis.get_csv_files(path)

df = None

for file in files:
    # Separa o nome da aplicação que está presente
    # no nome dos arquivos.
    tokens = file.split('-')
    tokens = tokens[-1].split('.pcap')
    if isinstance(df, pd.core.frame.DataFrame):
        # Lê o arquivo csv e armazena em um DataFrame
        new_df = pd.read_csv(file, delimiter=',')
        # Inclui o nome da aplicação na coluna NOME_APP
        new_df['NOME_APP'] = tokens[0]
        df = pd.concat([df, new_df])
    else:
        df = pd.read_csv(file, delimiter=',')
        df['NOME_APP'] = tokens[0]

df

```

```

[5]: print('Número de exemplos:', len(df))
print('Número de colunas:', len(df.columns))
print('Nomes das colunas:', [coluna for coluna in df.columns])

```

Número de exemplos: 2616579

Número de colunas: 86

Nomes das colunas: ['Flow ID', ' Source IP', ' Source Port', ' Destination IP',
' Destination Port', ' Protocol', ' Timestamp', ' Flow Duration', ' Total Fwd
Packets', ' Total Backward Packets', 'Total Length of Fwd Packets', ' Total
Length of Bwd Packets', ' Fwd Packet Length Max', ' Fwd Packet Length Min', '
Fwd Packet Length Mean', ' Fwd Packet Length Std', 'Bwd Packet Length Max', '
Bwd Packet Length Min', ' Bwd Packet Length Mean', ' Bwd Packet Length Std',

```
'Flow Bytes/s', ' Flow Packets/s', ' Flow IAT Mean', ' Flow IAT Std', ' Flow IAT
Max', ' Flow IAT Min', 'Fwd IAT Total', ' Fwd IAT Mean', ' Fwd IAT Std', ' Fwd
IAT Max', ' Fwd IAT Min', 'Bwd IAT Total', ' Bwd IAT Mean', ' Bwd IAT Std', '
Bwd IAT Max', ' Bwd IAT Min', 'Fwd PSH Flags', ' Bwd PSH Flags', ' Fwd URG
Flags', ' Bwd URG Flags', ' Fwd Header Length', ' Bwd Header Length', 'Fwd
Packets/s', ' Bwd Packets/s', ' Min Packet Length', ' Max Packet Length', '
Packet Length Mean', ' Packet Length Std', ' Packet Length Variance', 'FIN Flag
Count', ' SYN Flag Count', ' RST Flag Count', ' PSH Flag Count', ' ACK Flag
Count', ' URG Flag Count', ' CWE Flag Count', ' ECE Flag Count', ' Down/Up
Ratio', ' Average Packet Size', ' Avg Fwd Segment Size', ' Avg Bwd Segment
Size', ' Fwd Header Length.1', 'Fwd Avg Bytes/Bulk', ' Fwd Avg Packets/Bulk', '
Fwd Avg Bulk Rate', ' Bwd Avg Bytes/Bulk', ' Bwd Avg Packets/Bulk', 'Bwd Avg
Bulk Rate', 'Subflow Fwd Packets', ' Subflow Fwd Bytes', ' Subflow Bwd Packets',
' Subflow Bwd Bytes', 'Init_Win_bytes_forward', ' Init_Win_bytes_backward', '
act_data_pkt_fwd', ' min_seg_size_forward', 'Active Mean', ' Active Std', '
Active Max', ' Active Min', 'Idle Mean', ' Idle Std', ' Idle Max', ' Idle Min',
' Label', 'NOME_APP']
```

2.1 Amostragem dos dados

- Devido a grande quantidade de exemplos, mais de 2,5 milhões, a velocidade dos treinamentos pode ser demasiadamente lenta, portanto, neste trabalho será utilizado 3% do total de exemplos, resultando em um pouco mais de 78 mil exemplos.

```
[6]: # Descomentar
df = df.sample(frac=0.03, random_state=0)
print('Número de exemplos:', len(df))
```

Número de exemplos: 78497

```
[7]: # Distribuição das classes
df.groupby([' Label']).size()
```

```
[7]: Label
ADWARE_DOWGIN      1152
ADWARE_EWIND       1315
ADWARE_FEIWO       1623
ADWARE_GOOLIGAN    2865
ADWARE_KEMOGE      1182
ADWARE_KODOUS       959
ADWARE_MOBIDASH     951
ADWARE_SELFMITE     418
ADWARE_SHUANET     1221
ADWARE_YOUMI       1028
BENIGN             36344
MALWARE             84
RANSOMWARE_CHARGER 1206
RANSOMWARE_JISUT   789
```

RANSOMWARE_KOLER	1284
RANSOMWARE_LOCKERPIN	755
RANSOMWARE_PLETOR	132
RANSOMWARE_PORNDROID	1354
RANSOMWARE_RANSOMBO	1164
RANSOMWARE_SIMPLOCKER	1035
RANSOMWARE_SVPENG	1661
RANSOMWARE_WANNALOCKER	1023
SCAREWARE	229
SCAREWARE_ANDROIDDEFENDER	1700
SCAREWARE_ANDROIDSPY	768
SCAREWARE_AVFORANDROID	1219
SCAREWARE_AVPASS	1207
SCAREWARE_FAKEAPP	1036
SCAREWARE_FAKEAPPAL	1361
SCAREWARE_FAKEAV	1181
SCAREWARE_FAKEJOB OFFER	963
SCAREWARE_FAKETAOBAO	1047
SCAREWARE_PENETHO	688
SCAREWARE_VIRUSSHIELD	717
SMSMALWARE_BEANBOT	354
SMSMALWARE_BIIGE	970
SMSMALWARE_FAKEINST	459
SMSMALWARE_FAKEMART	173
SMSMALWARE_FAKENOTIFY	634
SMSMALWARE_JIFAKE	174
SMSMALWARE_MAZARBOT	178
SMSMALWARE_NANDROBOX	1361
SMSMALWARE_PLANKTON	1161
SMSMALWARE_SMSSNIFFER	1053
SMSMALWARE_ZZONE	317

dtype: int64

2.2 Ajuste na coluna *Label*

- Como o objetivo é prever *Malwares* para as categorias *Adware*, *Ransomware*, *Scareware* e *SMS Malware*, as famílias serão retiradas da coluna *Label*, mantendo apenas as categorias.

```
[8]: # Divide Label em Label e Family
df[['Label', 'Family']] = df['Label'].str.split('_', 1, expand=True)
# Remove a antiga coluna 'Label'
df = df.drop(columns=['Label'])
```

2.3 Remoção de colunas

- Serão removidas as colunas:
 - *Timestamp* - representa apenas a data e a hora em que os dados foram coletados, não

tendo relevância para o objetivo estabelecido.

- *Flow ID* - representa apenas um identificador do exemplo, não representando dado relevante para o treinamento.
- *Family* - Como o objetivo é apenas prever as categorias dos *Malwares* a Família será descartada.
- *Source IP* e *Destination IP* - Como os dados são de fluxo de tráfego de rede, e a captura se dá pelas duas vias, o IP do aparelho se repete muitas vezes em uma das duas colunas em todos os exemplos, e é provável que exemplos novos não possuam o mesmo IP do aparelho utilizado na captura, por este motivo essas duas colunas serão removidas.
- A coluna *NOME_APP* será mantida pois será utilizada mais para frente.

```
[9]: # Remove colunas desnecessárias
columns = list(df.columns)
columns.remove('Flow ID')
columns.remove('Source IP')
columns.remove('Destination IP')
columns.remove('Family')
columns.remove('Timestamp')
df = df[columns]
print('Número de colunas:', len(df.columns))
```

Número de colunas: 82

2.4 Remoção de exemplos

- Existe um subconjunto de exemplos rotulados apenas como *Malware*, como eles não possuem uma categoria serão eliminados.
- Também serão eliminados os registros que possuem exemplos nulos.

```
[10]: print('Número de exemplos:', len(df))
```

Número de exemplos: 78497

```
[11]: # Remove exemplos que têm o rótulo somente MALWARE
df = df[df['Label'] != 'MALWARE']
# Remove exemplos com valores nulos.
df = df.dropna(how='any', axis=0)
```

```
[12]: print('Número de exemplos:', len(df))
```

Número de exemplos: 78411

```
[13]: # Exibe DataFrame resultante
df
```

```
[13]:      Source Port  Destination Port  Protocol  Flow Duration \
311999      49738.0           443.0         6.0      314154.0
2315643     33281.0           443.0         6.0      183700.0
```

1348860	54449.0	5222.0	6.0	119979513.0
289365	55276.0	80.0	6.0	9791581.0
2106238	33483.0	443.0	6.0	117446695.0
...
644643	45990.0	80.0	6.0	31756458.0
1265866	58928.0	80.0	6.0	115910.0
2370110	39152.0	53.0	17.0	1414090.0
955043	37919.0	80.0	6.0	10552731.0
1045533	40188.0	80.0	6.0	785321.0

	Total Fwd Packets	Total Backward Packets \
311999	15.0	18.0
2315643	2.0	0.0
1348860	49.0	43.0
289365	2.0	0.0
2106238	23.0	16.0
...
644643	2.0	0.0
1265866	66.0	65.0
2370110	1.0	1.0
955043	2.0	2.0
1045533	12.0	11.0

	Total Length of Fwd Packets	Total Length of Bwd Packets \
311999	1637.0	19098.0
2315643	0.0	0.0
1348860	1111.0	1397.0
289365	0.0	0.0
2106238	4910.0	9771.0
...
644643	0.0	0.0
1265866	372.0	89571.0
2370110	32.0	207.0
955043	0.0	0.0
1045533	372.0	11524.0

	Fwd Packet Length Max	Fwd Packet Length Min	...	Active Mean \
311999	737.0	0.0	...	0.000000e+00
2315643	0.0	0.0	...	0.000000e+00
1348860	273.0	0.0	...	1.985529e+06
289365	0.0	0.0	...	0.000000e+00
2106238	541.0	0.0	...	2.683545e+06
...
644643	0.0	0.0	...	0.000000e+00
1265866	372.0	0.0	...	0.000000e+00
2370110	32.0	32.0	...	0.000000e+00
955043	0.0	0.0	...	0.000000e+00

1045533		372.0		0.0	...	0.000000e+00
---------	--	-------	--	-----	-----	--------------

	Active Std	Active Max	Active Min	Idle Mean	Idle Std	\
311999	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
2315643	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
1348860	2.351435e+06	6009953.0	194744.0	1.513963e+07	3.611158e+06	
289365	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
2106238	3.619360e+06	5242819.0	124271.0	5.602101e+07	6.875939e+07	
...	
644643	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
1265866	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
2370110	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
955043	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	
1045533	0.000000e+00	0.0	0.0	0.000000e+00	0.000000e+00	

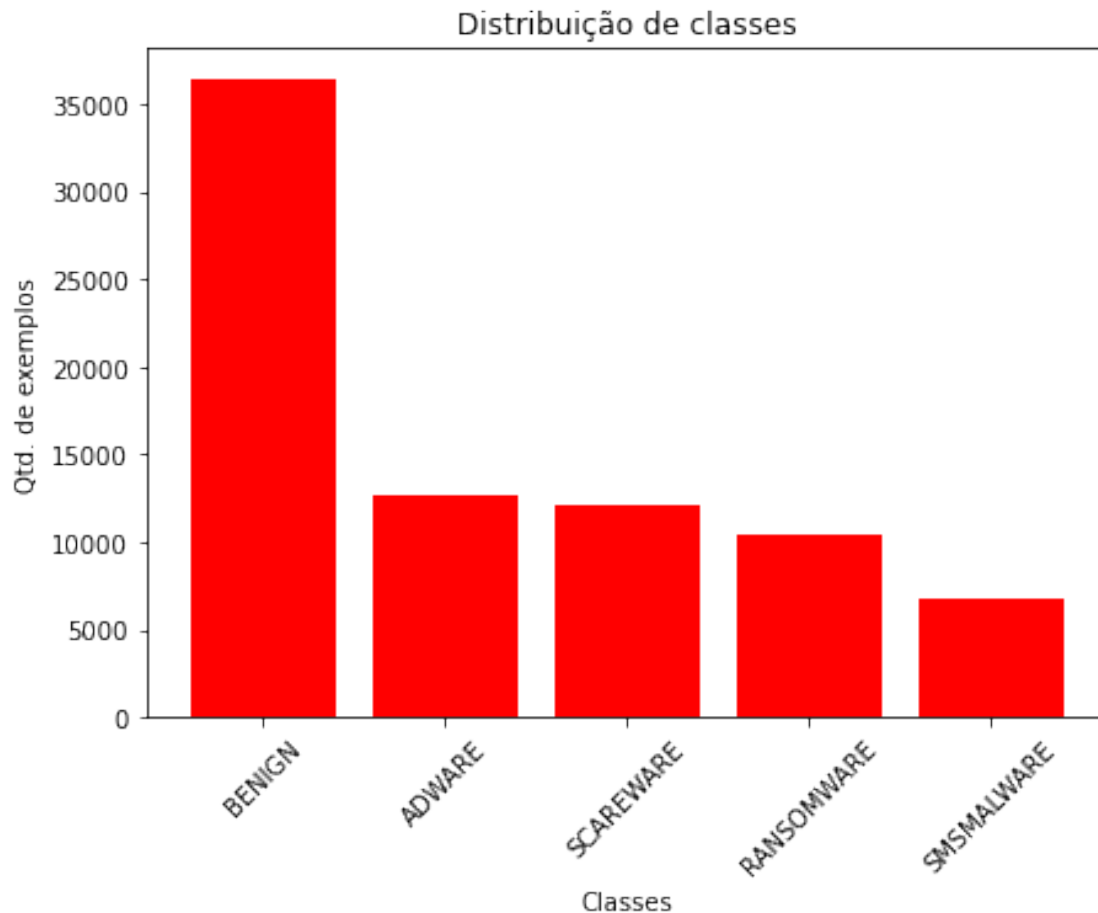
	Idle Max	Idle Min	NOME_APP	\
311999	0.0	0.0	2c7b372254730bfa366e4759db34812c	
2315643	0.0	0.0	com.asdhillon.admin.gymapplication	
1348860	20168014.0	10341591.0	68c9b7be13a2baef87370d291a5190ac	
289365	0.0	0.0	244c982e91b55edd55fab1c96c392211	
2106238	104641248.0	7400781.0	13bca905ccc119771040fd5fd30afff1	
...	
644643	0.0	0.0	zok.android.numbers	
1265866	0.0	0.0	b3a694ab3b58de4a944a3d4a03f67c6d	
2370110	0.0	0.0	com.bf.app7418de	
955043	0.0	0.0	com.jetappfactory.jetaudio	
1045533	0.0	0.0	br.com.tapps.deardiary	

	Label
311999	RANSOMWARE
2315643	BENIGN
1348860	ADWARE
289365	RANSOMWARE
2106238	SCAREWARE
...	...
644643	BENIGN
1265866	ADWARE
2370110	BENIGN
955043	BENIGN
1045533	BENIGN

[78411 rows x 82 columns]

2.5 Distribuição das amostras por classe

```
[14]: # Distribuição das classes
funcoes_uteis.plot_distribuicao_classes(df, 'Label')
```



2.6 Salvar *dataset* preparado

- Salvar dados preparados para uso futuro.

```
[15]: # Salva DataFrame em arquivo CSV
df.to_csv(path + 'dados_fluxo_trafego.csv', index=False)
# Reseta os índices
df.reset_index(drop=True, inplace=True)
```

```
[16]: # Abre arquivo
#path = '/home/efbaro/Documentos/dados/CSVs/'
#df = pd.read_csv(path + 'dados_fluxo_trafego.csv', delimiter=',')
```


2.7 Lendo e transformando dados de permissões dos Android_manifests

- Para enriquecer o *dataset* foi extraído das APKs relacionados aos dados os arquivos AndroidManifests e deles extraiu-se as permissões das aplicações, que foram transformadas em arquivos CSVs. Essa extração foi realizada com a ajuda do script [manifest_script.py](#). Os arquivos CSVs foram organizados em pastas da mesma forma que os CSVs dos dados de tráfego de rede.
- Esses dados foram concatenados aos dados de tráfego por meio do nome da aplicação.

```
[127]: # Lê as permissões em todos os arquivos e os coloca em um único dataset
path = '/home/efbaro/Documentos/dados/CSVs_permissions/'
files = funcoes_uteis.get_csv_files(path)

df_permissions = None

for i in range(len(files)):

    # Obtém rótulo a partir do diretório da aplicação
    label = files[i].replace(path, '')
    label = label.split('/')
    label = label[0].split('_')
    label = label[0]
    label

    if i == 0:
        df_permissions = pd.read_csv(files[i], delimiter=',')
        df_permissions['Label'] = label
    else:
        new_df = pd.read_csv(files[i], delimiter=',')
        new_df['Label'] = label
        df_permissions = pd.concat([df_permissions, new_df])

df_permissions
```

```
[127]:
```

	NOME_APP \
0	0639a74f508591f99a7d2309f5825fea
1	0bccdcc7d63f0754e9e66c806e8e9203
2	0df8e91d4ee84180099a26d97cf5baf4
3	00357b0e208c20df3182d54cb2ba15bf
4	02548535ff1cc285fddf699f2d77bcba
...	...
4	c57194d05a30d53c764983c70e471791
5	8be7ac1e01b3a5db14103187232d4f75
6	eda9098498a6201383e311f9b3757b4f
7	5d7af62c7f33522a8fe50085204d90c8
8	1c0c1837e99107f137c47e87570be00f

PERMISSOES	Label
------------	-------

0	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE CHANGE_...	Ransomware
1	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE CHANGE_...	Ransomware
2	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE CHANGE_...	Ransomware
3	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE CHANGE_...	Ransomware
4	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE CALL_PH...	Ransomware
..
4	ACCESS_NETWORK_STATE INTERNET READ_PHONE_STATE...	Scareware
5	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION AC...	Scareware
6	GET_TASKS RECEIVE_BOOT_COMPLETED RECEIVE_SMS R...	Scareware
7	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE BROADCA...	Scareware
8	ACCESS_NETWORK_STATE ACCESS_WIFI_STATE BROADCA...	Scareware

[2115 rows x 3 columns]

```
[128]: # Remove exemplos que não tem permissões
df_permissions = df_permissions[~df_permissions['PERMISSOES'].isnull()]
```

2.7.1 Conversão da coluna de permissões em várias colunas com as respectivas permissões

```
[129]: # Cria uma lista com as permissões
permlist = df_permissions.values.tolist()

big_list = []
for l in permlist:
    big_list += [[l[0]] + l[1].split('|') + [l[2]]]
len(big_list)
```

[129]: 2067

```
[130]: # Obtenção do espaço amostral das permissões únicas
uniqueperms = {x for l in big_list for x in l[1:-1]}
print(len(uniqueperms))
```

469

```
[131]: # Definindo o novo cabeçalho
df_header = ['NOME_APP'] + list(uniqueperms) + ['Label']
len(df_header)
```

[131]: 471

```
[132]: # Preenchimento do dataframe de treino
df_train_content = []
nperm = len(df_header)
for sample in big_list:
    has_permission = [0] * nperm
```

```

has_permission[0] = sample[0]
has_permission[-1] = sample[-1]
for perm in sample[1:-1]:
    has_permission[df_header.index(perm)] = 1
df_train_content.append(has_permission)
df_permissoes_train = pd.DataFrame(df_train_content, columns=df_header)
pd.set_option('display.max_rows', 150)
df_permissoes_train

```

[132]:

	NOME_APP	SENSOR_ENABLE	GALLERY_PROVIDER	\
0	0639a74f508591f99a7d2309f5825fea	0	0	
1	0bccdcc7d63f0754e9e66c806e8e9203	0	0	
2	0df8e91d4ee84180099a26d97cf5baf4	0	0	
3	00357b0e208c20df3182d54cb2ba15bf	0	0	
4	02548535ff1cc285fddf699f2d77bcba	0	0	
...	
2062	c57194d05a30d53c764983c70e471791	0	0	
2063	8be7ac1e01b3a5db14103187232d4f75	0	0	
2064	eda9098498a6201383e311f9b3757b4f	0	0	
2065	5d7af62c7f33522a8fe50085204d90c8	0	0	
2066	1c0c1837e99107f137c47e87570be00f	0	0	

	CEMOJI	WRITE_HISTORY_BOOKMARKS	lh2	NFC_SE	AUTHENTICATE_ACCOUNTS	\
0	0	0	0	0	0	
1	0	0	0	0	0	
2	0	0	0	0	0	
3	0	0	0	0	0	
4	0	0	0	0	0	
...	
2062	0	0	0	0	0	
2063	0	0	0	0	0	
2064	0	0	0	0	0	
2065	0	0	0	0	0	
2066	0	0	0	0	0	

	DELETE_CACHE_FILES	MDM_CONTENT_PROVIDER	...	DISABLE_KEYGUARD	wise	\
0	0	0	...	0	0	
1	0	0	...	0	0	
2	0	0	...	0	0	
3	0	0	...	0	0	
4	0	0	...	0	0	
...	
2062	0	0	...	0	0	
2063	0	0	...	0	0	
2064	0	0	...	0	0	
2065	0	0	...	1	0	
2066	0	0	...	1	0	

	RECORD_VIDEO	MODIFY_PHONE_STATE	ACTION_BOOT_COMPLETED	\
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0	0	
...	
2062	0	0	0	
2063	1	0	0	
2064	0	0	0	
2065	0	0	0	
2066	0	0	0	

	AVIATE_INTER_APP	jotspot	SMARTCARD	sierraga	Label
0	0	0	0	0	Ransomware
1	0	0	0	0	Ransomware
2	0	0	0	0	Ransomware
3	0	0	0	0	Ransomware
4	0	0	0	0	Ransomware
...	
2062	0	0	0	0	Scareware
2063	0	0	0	0	Scareware
2064	0	0	0	0	Scareware
2065	0	0	0	0	Scareware
2066	0	0	0	0	Scareware

[2067 rows x 471 columns]

```
[133]: # Salva dados de permissões transformados
df_permissoes_train.to_csv(path+ 'permissoes_train.csv', index=False)
# Reseta os índices
df_permissoes_train.reset_index(drop=True, inplace=True)
```

```
[134]: # Abre arquivo
#path = '/home/efbaro/Documentos/dados/CSVs_permissions/'
#df_permissoes_train = pd.read_csv(path + 'permissoes_train.csv', delimiter=',')
```

```
[135]: # Remove label das permissoes
columns = list(df_permissoes_train.columns)
columns.remove('Label')
df_permissoes_train = df_permissoes_train[columns]
```

2.7.2 Junção dos datasets

```
[136]: # Faz a junção das permissões com os dados de fluxo de trafego
df = pd.merge(df, df_permissoes_train, how="left", on='NOME_APP')
```

```
[137]: # Muda posição da coluna Label para última posição do DataFrame
colunas = list(df.columns)
colunas.remove('Label')
colunas.append('Label')
df = df[colunas]
df
```

```
[137]:
```

	Source Port	Destination Port	Protocol	Flow Duration	\
0	49738.0	443.0	6.0	314154.0	
1	33281.0	443.0	6.0	183700.0	
2	54449.0	5222.0	6.0	119979513.0	
3	55276.0	80.0	6.0	9791581.0	
4	33483.0	443.0	6.0	117446695.0	
...	
80694	45990.0	80.0	6.0	31756458.0	
80695	58928.0	80.0	6.0	115910.0	
80696	39152.0	53.0	17.0	1414090.0	
80697	37919.0	80.0	6.0	10552731.0	
80698	40188.0	80.0	6.0	785321.0	

	Total Fwd Packets	Total Backward Packets	\
0	15.0	18.0	
1	2.0	0.0	
2	49.0	43.0	
3	2.0	0.0	
4	23.0	16.0	
...	
80694	2.0	0.0	
80695	66.0	65.0	
80696	1.0	1.0	
80697	2.0	2.0	
80698	12.0	11.0	

	Total Length of Fwd Packets	Total Length of Bwd Packets	\
0	1637.0	19098.0	
1	0.0	0.0	
2	1111.0	1397.0	
3	0.0	0.0	
4	4910.0	9771.0	
...	
80694	0.0	0.0	
80695	372.0	89571.0	

80696	32.0	207.0
80697	0.0	0.0
80698	372.0	11524.0

	Fwd Packet Length Max	Fwd Packet Length Min	...	DISABLE_KEYGUARD	\
0	737.0	0.0	...	0.0	
1	0.0	0.0	...	0.0	
2	273.0	0.0	...	0.0	
3	0.0	0.0	...	0.0	
4	541.0	0.0	...	0.0	
...	
80694	0.0	0.0	...	0.0	
80695	372.0	0.0	...	0.0	
80696	32.0	32.0	...	0.0	
80697	0.0	0.0	...	1.0	
80698	372.0	0.0	...	0.0	

	wise	RECORD_VIDEO	MODIFY_PHONE_STATE	ACTION_BOOT_COMPLETED	\
0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	
...	
80694	0.0	0.0	0.0	0.0	
80695	0.0	0.0	0.0	0.0	
80696	0.0	1.0	0.0	0.0	
80697	0.0	0.0	0.0	0.0	
80698	0.0	0.0	0.0	0.0	

	AVIATE_INTER_APP	jotspot	SMARTCARD	sierraqa	Label
0	0.0	0.0	0.0	0.0	RANSOMWARE
1	0.0	0.0	0.0	0.0	BENIGN
2	0.0	0.0	0.0	0.0	ADWARE
3	0.0	0.0	0.0	0.0	RANSOMWARE
4	0.0	0.0	0.0	0.0	SCAREWARE
...	
80694	0.0	0.0	0.0	0.0	BENIGN
80695	0.0	0.0	0.0	0.0	ADWARE
80696	0.0	0.0	0.0	0.0	BENIGN
80697	0.0	0.0	0.0	0.0	BENIGN
80698	0.0	0.0	0.0	0.0	BENIGN

[80699 rows x 551 columns]

```
[138]: print('Número de exemplos:', len(df))
```

Número de exemplos: 80699

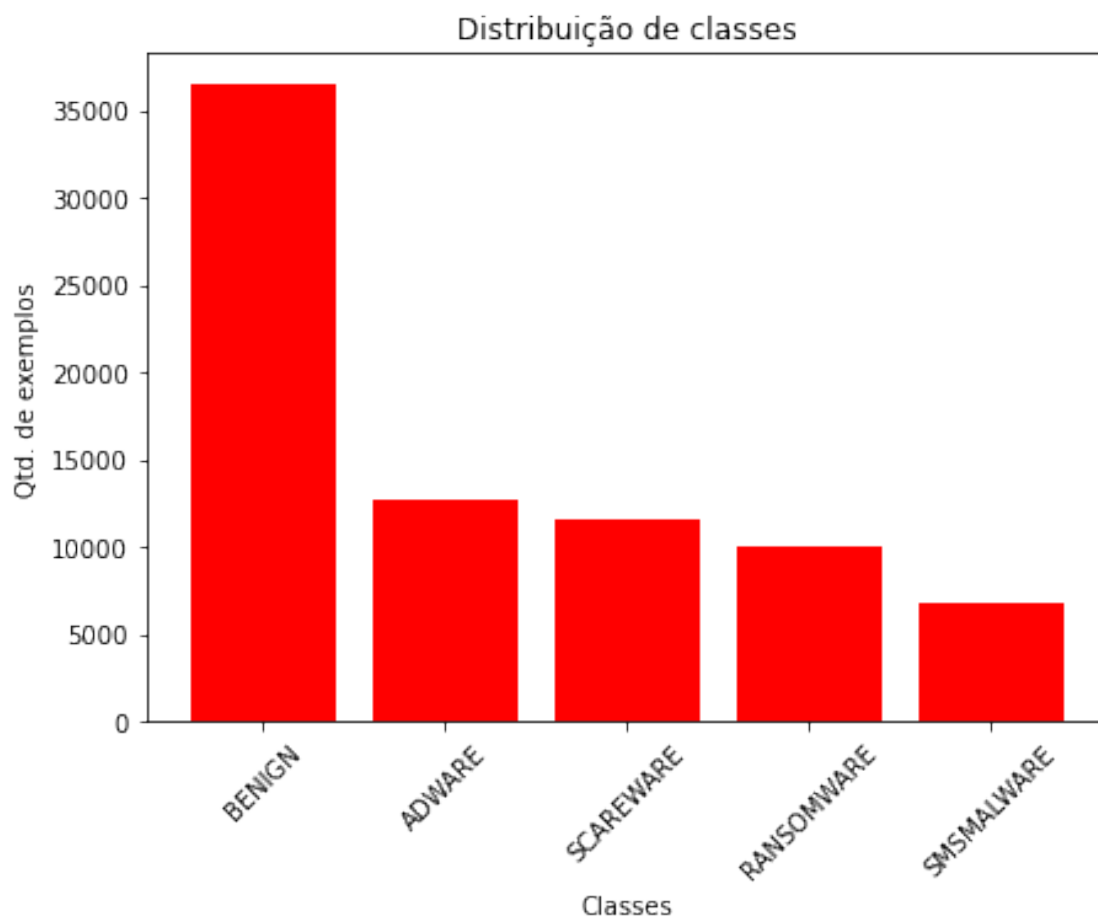
```
[139]: # Remove exemplos com valores nulos.  
df = df.dropna(how='any', axis=0)
```

```
[140]: print('Número de exemplos:', len(df))  
print('Número de colunas:', len(df.columns))
```

Número de exemplos: 77542

Número de colunas: 551

```
[141]: # Exibe novamente a distribuição dos dados  
funcoes_uteis.plot_distribuiacao_classes(df, 'Label')
```



3 Treinamento

- O treinamento será feito com os algoritmos KNN, Random Forest e MLP.

```
[142]: # Lista com os melhores classificados que serão utilizados na validação
dic_melhores_modelos = {'split_perc_knn': [],
                        'split_perc_rf': [],
                        'split_perc_mlp': [],
                        'melhor_val_cruz_knn': [],
                        'melhor_val_cruz_rf': [],
                        'melhor_val_cruz_mlp': []}
```

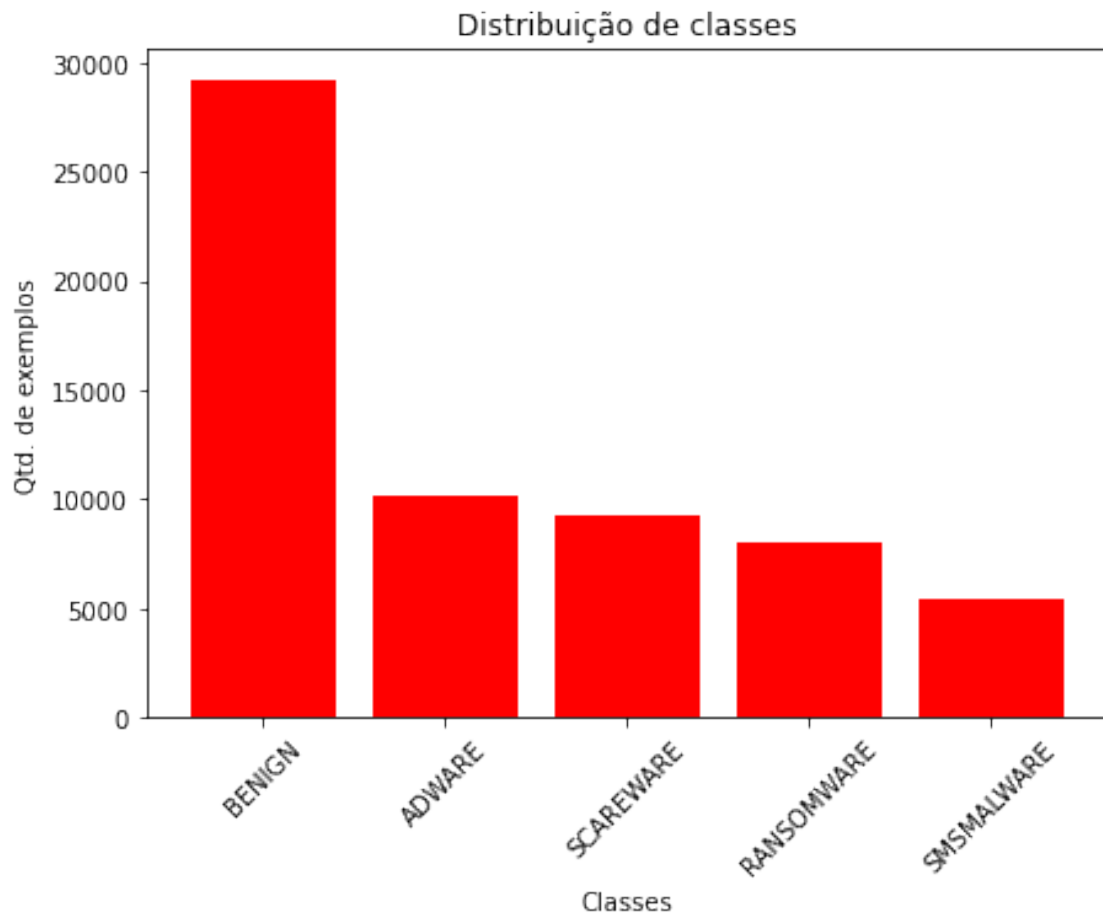
3.1 Divisão do *dataset*

- O *dataset* será dividido em duas porções uma de validação com 20% do total de dados, e outra para etapa de treinamento com os outros 80%.
- Na sequência os 80% para treinamento serão divididos em mais duas partes dos quais serão 80% para treinamento dos modelos e 20% para teste.
- A separação é feita misturando os dados (`shuffle=True`) e estratificada pela classe (`stratify=df['Label']`) buscando com isso manter a distribuição dos dados.

```
[143]: # Separação inicial 80/20
df_treinamento, df_validacao = train_test_split(df, test_size=0.2,
↪shuffle=True, stratify=df['Label'])
```

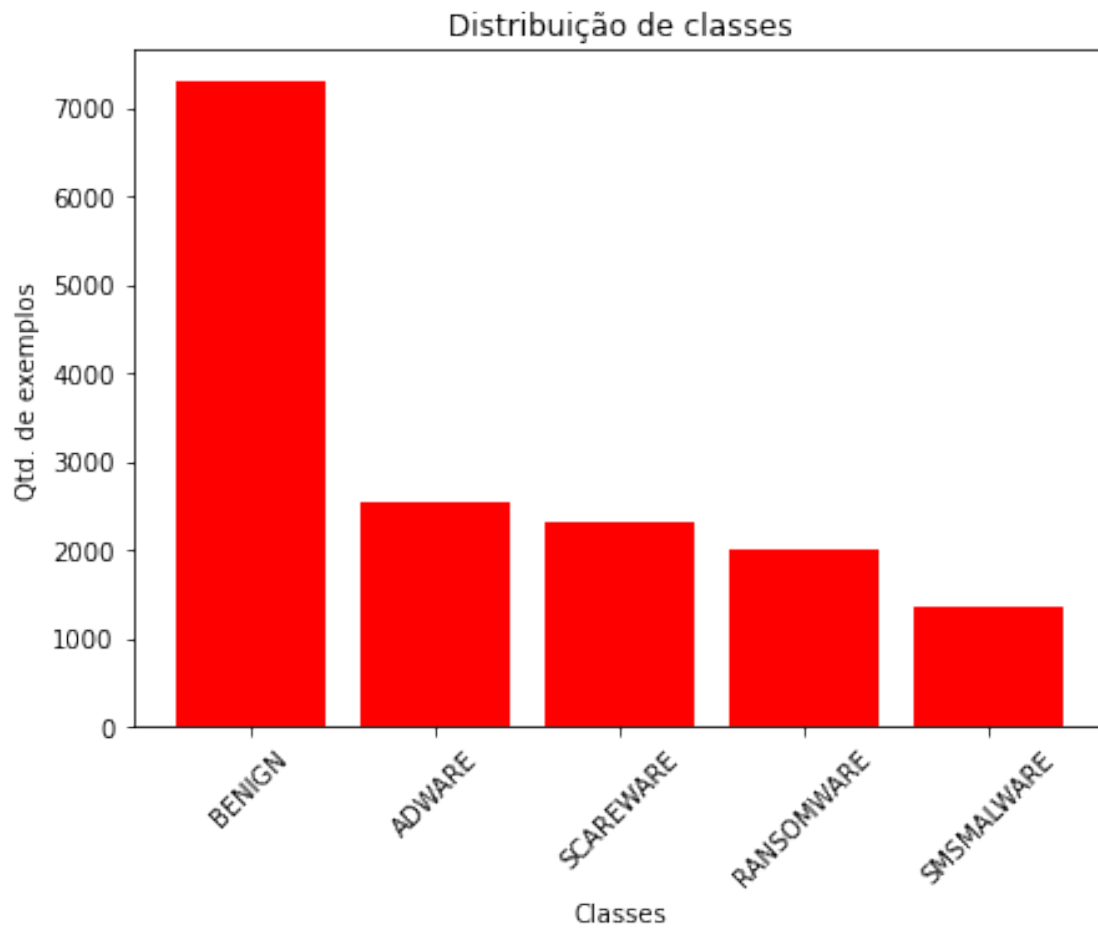
3.1.1 Distribuição das amostras de treinamento

```
[13]: funcoes_uteis.plot_distribuicao_classes(df_treinamento, 'Label')
```

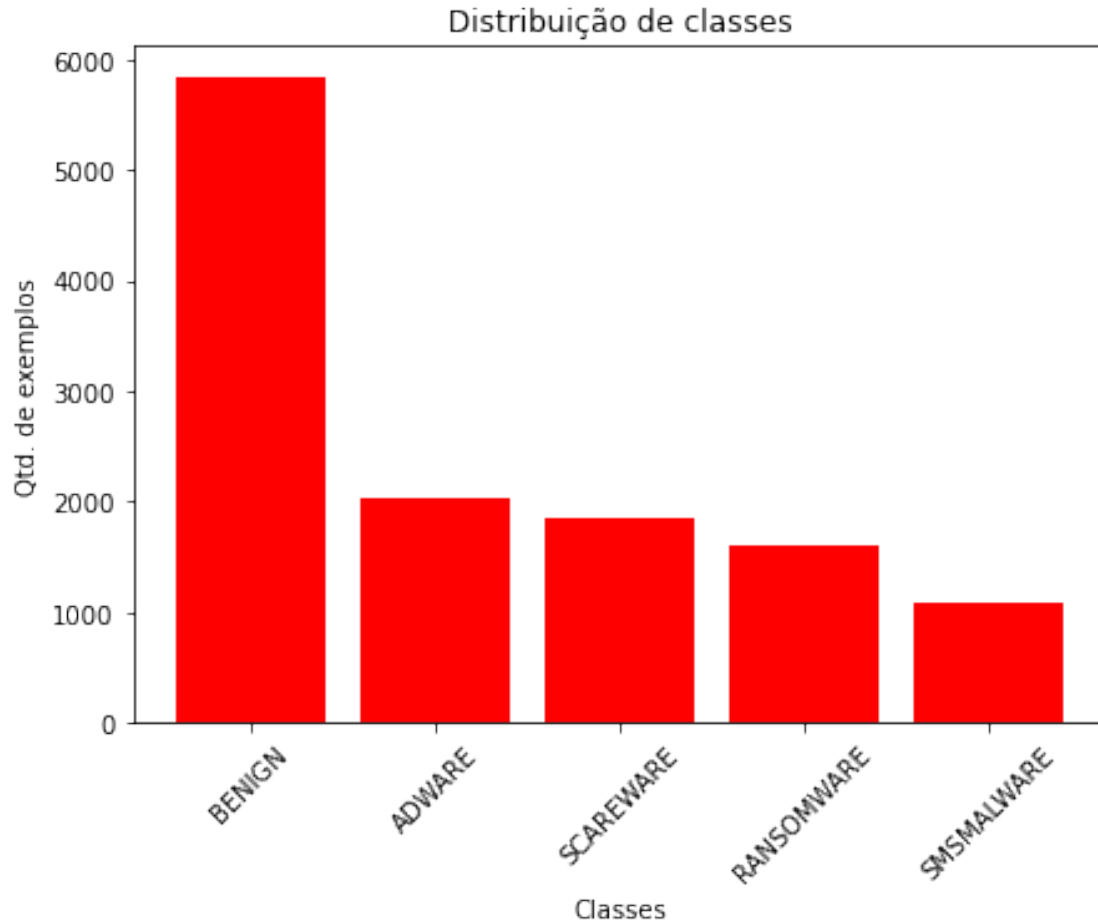
3.1.2 Distribuição das amostras de validação

```
[14]: funcoes_uteis.plot_distribuiacao_classes(df_validacao, 'Label')
```



```
[144]: # Separação para treinamento 80/20 sobre o df_treinamento
df_train, df_teste = train_test_split(df_treinamento, test_size=0.2,
↳ shuffle=True, stratify=df_treinamento['Label'])
```

```
[16]: funcoes_uteis.plot_distribuicao_classes(df_teste, 'Label')
```



3.2 Normalização

- Muitos atributos do *dataset* selecionado possuem diferença de escala muito grande, como os atributos '*Flow Duration*' e '*Total Fwd Packets*'. Essa diferença de escala, dependendo do classificador utilizado, pode dar uma significância muito maior para o primeiro "Flow Duration", já que ele varia muito mais em escala.
- Para resolver este problema será realizada a normalização dos atributos que não representam valores categóricos. No *dataset* utilizado apenas os atributos '*Source Port*', '*Destination Port*', '*Protocol*', '*Label*' e '*NOME_APP*' representam atributos categóricos, os demais atributos serão portanto normalizados.
- Para normalização será utilizado o [min-max scaler](#) do scikit learn. O *Min-max Scaling* reescala todas as características em um intervalo entre zero e um, utilizando a equação:

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

em que: - x_i é o valor da característica x de um exemplar i ; - $\min(x)$ é o menor valor da característica

x no conjunto de treino; $- \max(x)$ é o maior valor da característica x no conjunto de treino.

```
[145]: # Obtém colunas
colunas = list(df_train.columns)
# Colunas que devem ser removidas
colunas_remov = [' Source Port', ' Destination Port', ' Protocol', 'NOME_APP',
↳ 'Label']
# Remove colunas que NÃO serão normalizadas
for coluna in colunas_remov:
    colunas.remove(coluna)

# Inicializa o MinMax
scaler = MinMaxScaler(feature_range=(0,1))
# Treina MinMax somente as colunas selecionadas
scaler.fit(df_train[colunas])
# Transform as características
train_caract_norm = scaler.transform(df_train[colunas])
teste_caract_norm = scaler.transform(df_teste[colunas])
valid_caract_norm = scaler.transform(df_validacao[colunas])
treinamento_caract_norm = scaler.transform(df_treinamento[colunas])

# Cria os DataFrames normalizados
df_train_norm = pd.DataFrame(train_caract_norm, columns=colunas, index=df_train.
↳ index)
df_teste_norm = pd.DataFrame(teste_caract_norm, columns=colunas, index=df_teste.
↳ index)
df_validacao_norm = pd.DataFrame(valid_caract_norm,
↳ columns=colunas, index=df_validacao.index)
df_treinamento_norm = pd.DataFrame(treinamento_caract_norm,
↳ columns=colunas, index=df_treinamento.index)

# Concatena as colunas não normalizadas aos novos DataFrame e reorganiza as
↳ colunas
df_train_norm = pd.concat([df_train_norm, df_train[colunas_remov]], axis=1)
df_train_norm = df_train_norm[df_train.columns]

df_teste_norm = pd.concat([df_teste_norm, df_teste[colunas_remov]], axis=1)
df_teste_norm = df_teste_norm[df_teste.columns]

df_validacao_norm = pd.concat([df_validacao_norm,
↳ df_validacao[colunas_remov]], axis=1)
df_validacao_norm = df_validacao_norm[df_validacao.columns]

df_treinamento_norm = pd.concat([df_treinamento_norm,
↳ df_treinamento[colunas_remov]], axis=1)
df_treinamento_norm = df_treinamento_norm[df_treinamento.columns]
```

3.3 Treinamento *percentage split*

- Nesta etapa serão realizados treinamentos com a divisão dos dados em duas partes uma de 80% para treinamento e outra de 20%, essa divisão foi realizada sobre os 80% dos dados, como já visto nas etapas anteriores
- Inicialmente serão realizados os treinamentos variando alguns hiperparâmetros dos algoritmos para identificar os que proporcionam os melhores resultados, analisando para isso a acurácia de cada modelo gerado. Na sequência os melhores hiperparâmetros serão utilizados para um novo treinamento em que serão exibidos a matriz de confusão, Curva ROC, acurácia e erro do modelo.

3.3.1 Preparação dos dados para treinamento, normalizados e não normalizados

```
[146]: # Remove a coluna NOME_APP
df_train_norm = df_train_norm.drop('NOME_APP', axis=1)
df_train = df_train.drop('NOME_APP', axis=1)

df_teste_norm = df_teste_norm.drop('NOME_APP', axis=1)
df_teste = df_teste.drop('NOME_APP', axis=1)
```

```
[147]: # Dados de treinamento
X_train_norm = df_train_norm.iloc[:,0:-1].values
y_train_norm = df_train_norm.iloc[:,-1].values

X_train = df_train.iloc[:,0:-1].values
y_train = df_train.iloc[:,-1].values
```

```
[148]: # Dados de teste
X_teste_norm = df_teste_norm.iloc[:,0:-1].values
y_teste_norm = df_teste_norm.iloc[:,-1].values

X_teste = df_teste.iloc[:,0:-1].values
y_teste = df_teste.iloc[:,-1].values
```

3.3.2 KNN (*K-nearest neighbors algorithm*)

- Neste algoritmo serão variados o parâmetro número de vizinhos (*n_neighbors*) de 3 até 100 para detectar o melhor parâmetro. As métricas de distância testadas serão a *euclidean* e *manhattan*

```
[111]: numero_de_vizinhos = [3,5,7,9,11,13,15,25,45,70,100]
metricas_de_distancia = ['euclidean', 'manhattan']
```

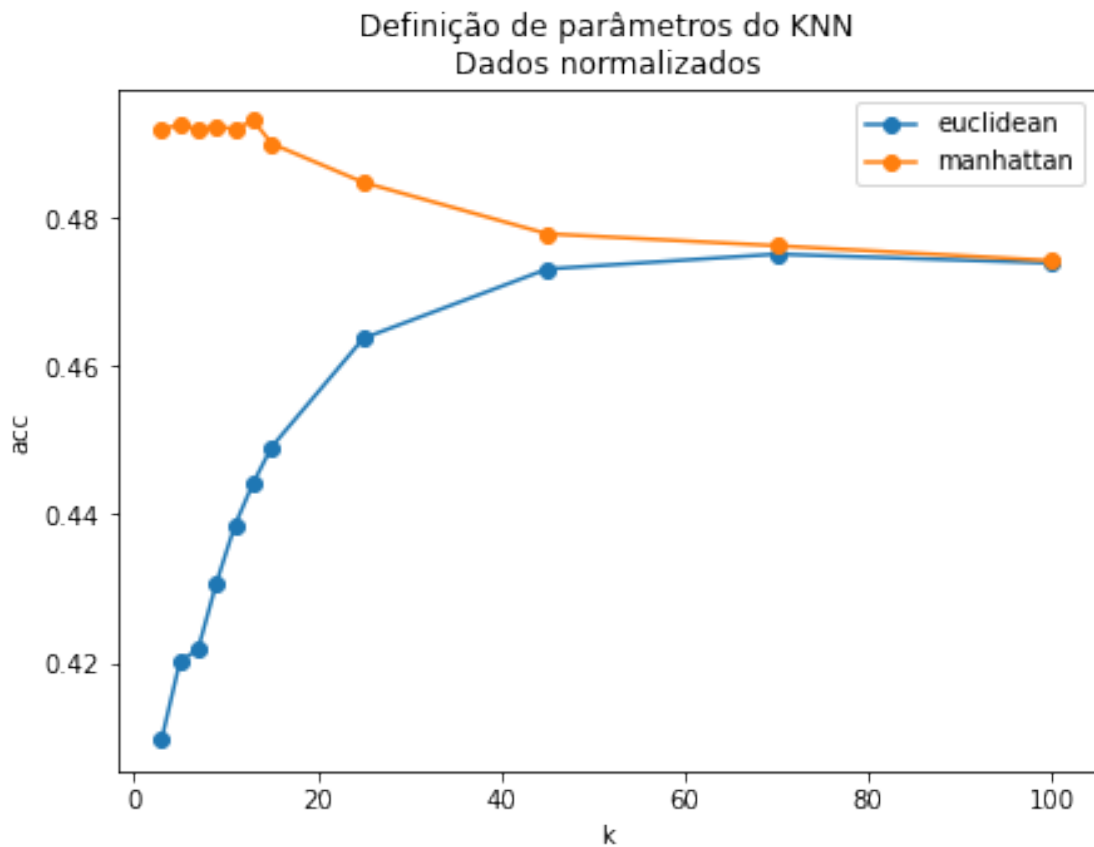
```
[112]: # Dados normalizados
acc = []
ks = []
metricas = []
for metrica in metricas_de_distancia:
```

```

for k in numero_de_vizinhos:
    # Cria modelo
    clf_KNN = KNeighborsClassifier(n_neighbors=k, metric=metrica)
    # Treina modelo
    clf_KNN.fit(X_train_norm, y_train_norm)
    # Faz predição sobre os dados de teste
    y_pred = clf_KNN.predict(X_teste_norm)
    # Calcula a acurácia
    acuracia = accuracy_score(y_teste_norm, y_pred)
    acc.append(acuracia)
    ks.append(k)
    metricas.append(metrica)

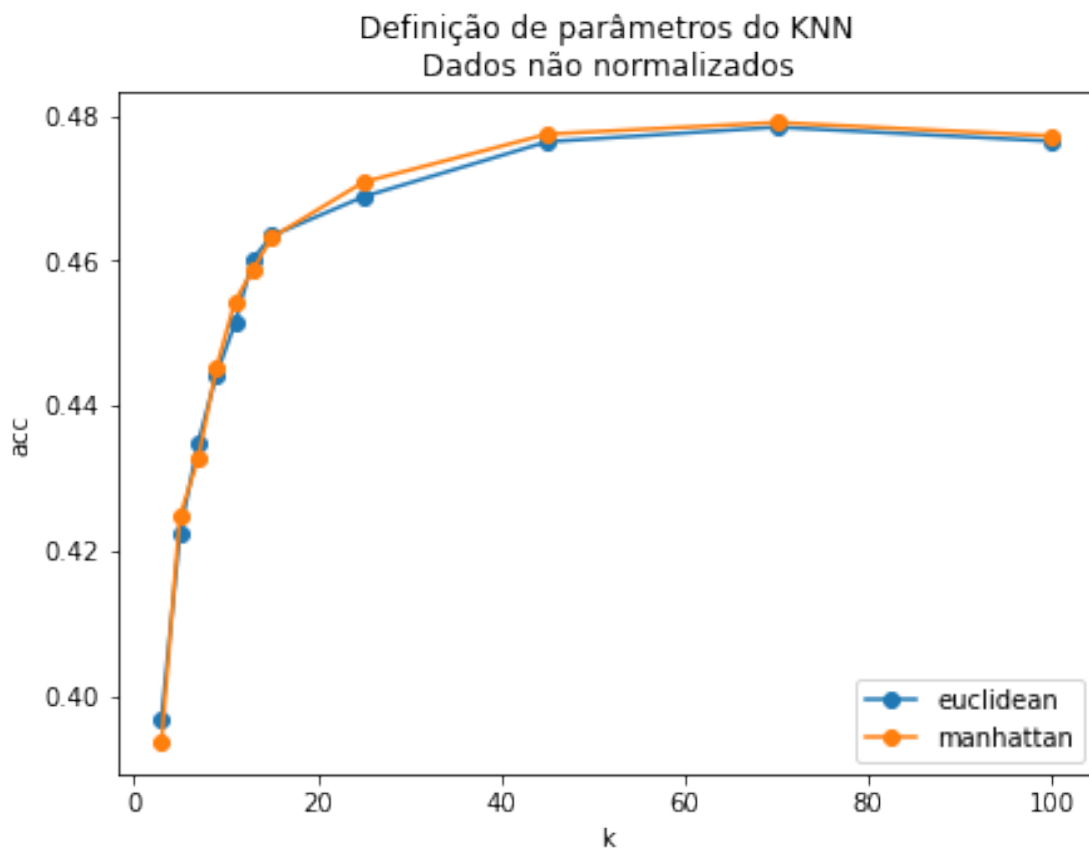
# Exibe os resultados
funcoes_uteis.plot_resultados(ks, 'k',
                              acc, 'acc', metricas,
                              'Definição de parâmetros do KNN\ndados_
↳normalizados')

```



```
[113]: # Dados não normalizados
acc = []
ks = []
metricas = []
for metrica in metricas_de_distancia:
    for k in numero_de_vizinhos:
        # Cria modelo
        clf_KNN = KNeighborsClassifier(n_neighbors=k, metric=metrica)
        # Treina modelo
        clf_KNN.fit(X_train, y_train)
        # Faz predição sobre os dados de teste
        y_pred = clf_KNN.predict(X_teste)
        # Calcula a acurácia
        acuracia = accuracy_score(y_teste, y_pred)
        acc.append(acuracia)
        ks.append(k)
        metricas.append(metrica)

# Exibe os resultados
funcoes_uteis.plot_resultados(ks, 'k',
                              acc, 'acc', metricas,
                              'Definição de parâmetros do KNN\nDados não normalizados')
```



Discussão

- No KNN a utilização dos **dados normalizados** chegaram a resultados sensivelmente melhores, pelos experimentos o melhor valor de k foi 13 e a melhor métrica de distância foi a *manhattan*. Portanto o experimento para o KNN será executado com os seguintes hiperparâmetros e dados:
 - Dados: Normalizados
 - Número de vizinhos (k): 13
 - Métrica de distância: *manhattan*

```
[87]: clf_KNN = KNeighborsClassifier(n_neighbors=13, metric='manhattan')
# Treina modelo
clf_KNN.fit(X_train_norm, y_train_norm)
# Faz predição sobre os dados de teste
y_pred = clf_KNN.predict(X_teste_norm)
# Imprime resultados
print(classification_report(y_pred, y_teste_norm, zero_division=True))
```

	precision	recall	f1-score	support
ADWARE	0.19	0.46	0.27	857
BENIGN	0.94	0.50	0.66	10821
RANSOMWARE	0.07	0.35	0.12	339
SCAREWARE	0.04	0.25	0.08	322
SMSMALWARE	0.02	0.37	0.04	68
accuracy			0.49	12407
macro avg	0.25	0.39	0.23	12407
weighted avg	0.83	0.49	0.60	12407

```
[88]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_teste, y_pred)
erro = 1 - accuracy
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

Acurácia: 0.490

Erro: 0.510

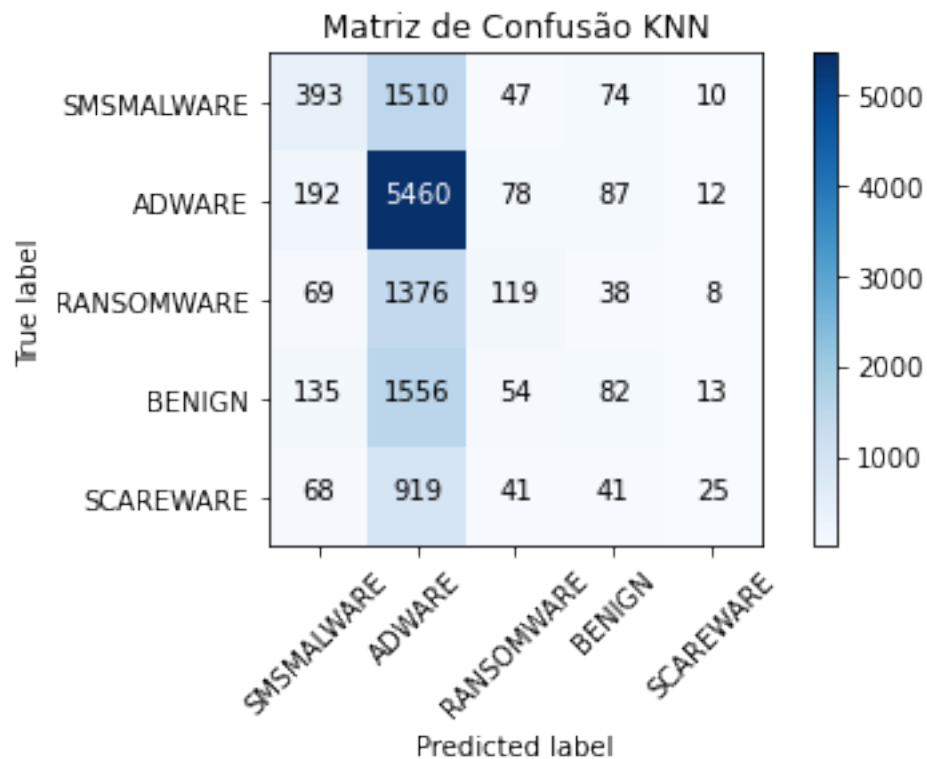
```
[89]: # Guarda o modelo gerado
dic_melhores_modelos['split_perc_knn'] = []
dic_melhores_modelos['split_perc_knn'] = []

dic_melhores_modelos['split_perc_knn'].append(clf_KNN)
```

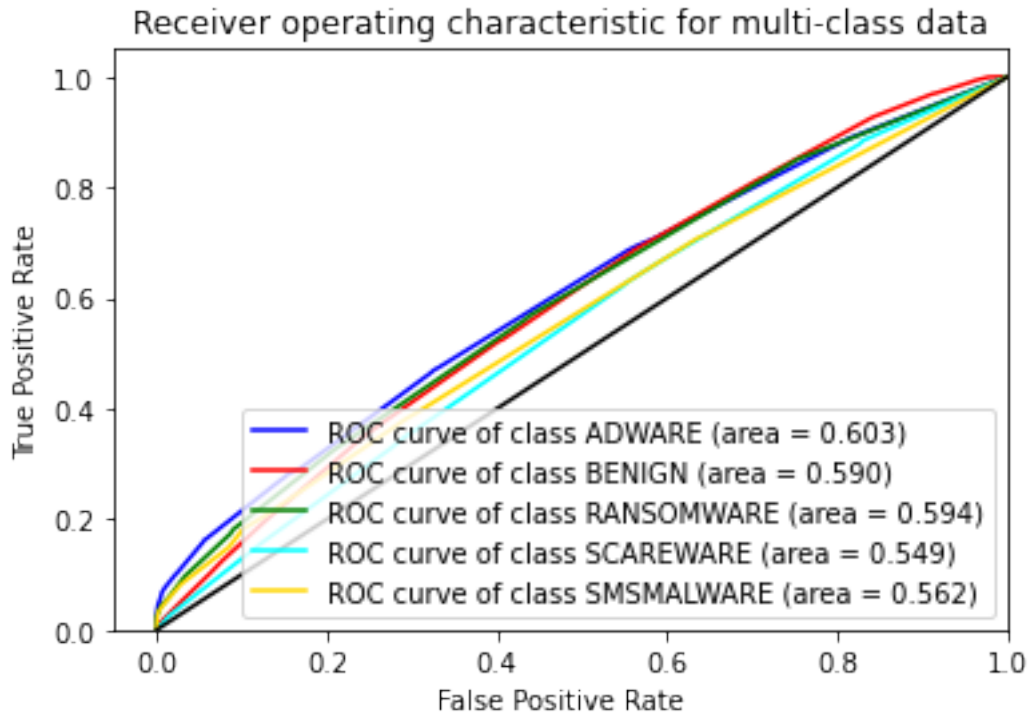


```
dic_melhores_modelos['split_perc_knn'].append(accuracy)
```

```
[90]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_teste, y_pred,
                                     set(y_teste),
                                     'Matriz de Confusão KNN')
```



```
[119]: # Exibe curva ROC
funcoes_uteis.plot_roc_curve(clf_KNN, X_train, y_train, X_teste, y_teste)
```



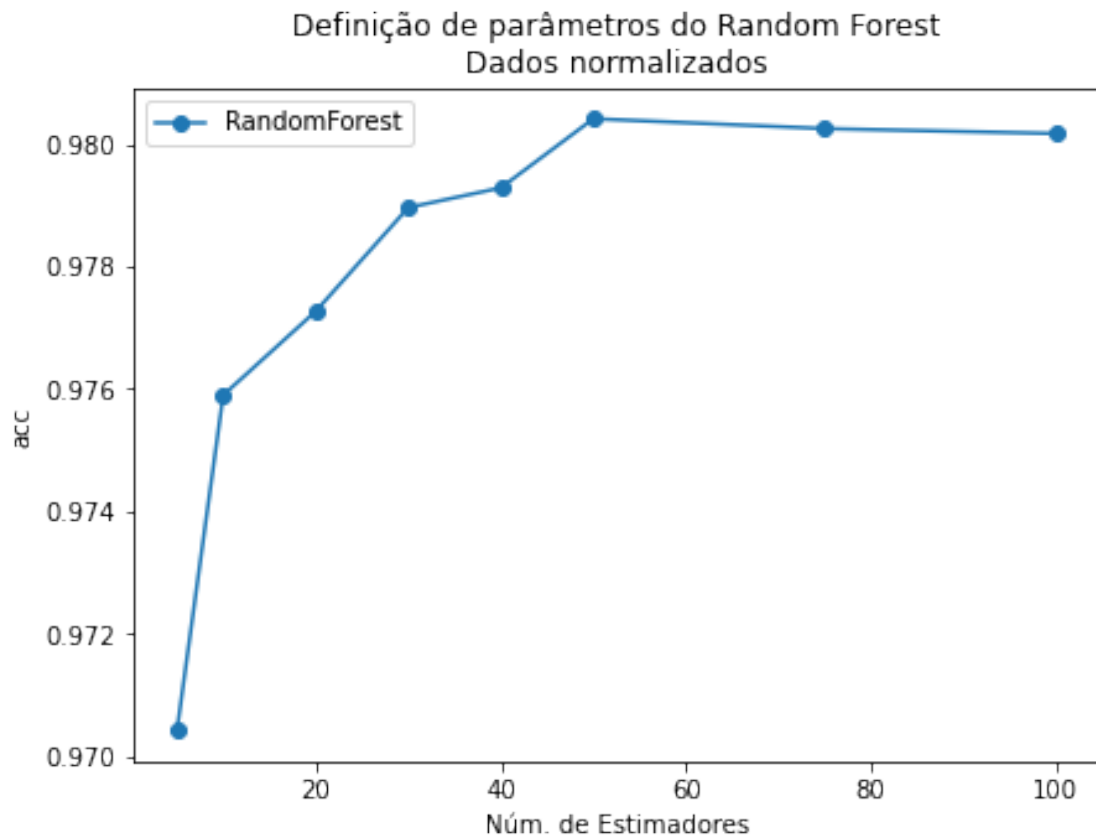
3.3.3 Random Forest

- Neste algoritmo foi feita a variação do hiperparâmetro número estimadores ($n_estimators$) de 5 até 100 para detectar o melhor.

```
[121]: numero_de_estimadores = [5,10,20,30,40,50,75,100]
```

```
[122]: # Dados normalizados
acc = []
num_estimadores = []
divisao = []
for n_estimators in numero_de_estimadores:
    # Cria modelo
    clf_rf = RandomForestClassifier(n_estimators=n_estimators, random_state=0)
    # Treina modelo
    clf_rf.fit(X_train_norm, y_train_norm)
    # Faz predição sobre os dados de teste
    y_pred = clf_rf.predict(X_teste_norm)
    # Calcula a acurácia
    acuracia = accuracy_score(y_teste_norm, y_pred)
    acc.append(acuracia)
    num_estimadores.append(n_estimators)
    divisao.append('RandomForest')
```

```
# Exibe os resultados
funcoes_uteis.plot_resultados(num_estimadores, 'Núm. de Estimadores',
                              acc, 'acc', divisao,
                              'Definição de parâmetros do Random Forest\nDados_
↪normalizados')
```



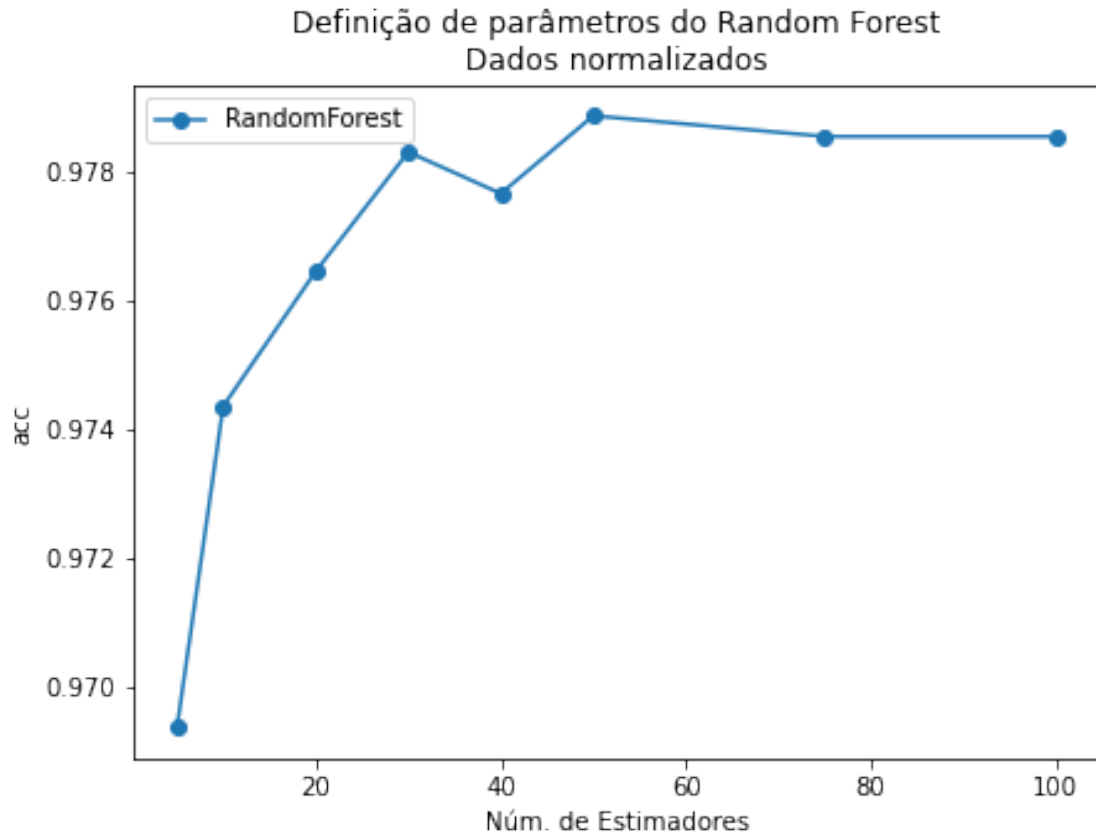
```
[123]: # Dados não normalizados
acc = []
num_estimadores = []
divisao = []
for n_estimators in numero_de_estimadores:
    # Cria modelo
    clf_rf = RandomForestClassifier(n_estimators=n_estimators, random_state=0)
    # Treina modelo
    clf_rf.fit(X_train, y_train)
    # Faz predição sobre os dados de teste
    y_pred = clf_rf.predict(X_teste)
    # Calcula a acurácia
    acuracia = accuracy_score(y_teste, y_pred)
    acc.append(acuracia)
```

```

num_estimadores.append(n_estimators)
divisao.append('RandomForest')

# Exibe os resultados
funcoes_uteis.plot_resultados(num_estimadores, 'Núm. de Estimadores',
                              acc, 'acc', divisao,
                              'Definição de parâmetros do Random Forest\nDados_
→normalizados')

```



Discussão

- Para o *RandomForest* a utilização dos **dados normalizados** chegaram a resultados sensivelmente melhores, pelos experimentos a melhor quantidade de estimadores foi 50. Portanto o experimento para o *RandomForest* será executado com os seguintes hiperparâmetros e dados:
 - Dados: Normalizados
 - Número de estimadores: 50

```

[91]: # Cria modelo
      clf_rf = RandomForestClassifier(n_estimators = 50, random_state=0)
      # Treina modelo

```

```

clf_rf.fit(X_train_norm, y_train_norm)
# Faz predição sobre os dados de teste
y_pred = clf_rf.predict(X_teste_norm)
# Imprime resultados
print(classification_report(y_pred, y_teste_norm, zero_division=True))

```

	precision	recall	f1-score	support
ADWARE	0.97	0.99	0.98	2000
BENIGN	1.00	0.97	0.98	5970
RANSOMWARE	0.98	1.00	0.99	1581
SCAREWARE	0.96	0.99	0.97	1775
SMSMALWARE	0.96	0.97	0.96	1081
accuracy			0.98	12407
macro avg	0.97	0.98	0.98	12407
weighted avg	0.98	0.98	0.98	12407

```

[92]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_teste_norm, y_pred)
erro = 1 - accuracy
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))

```

Acurácia: 0.981
 Erro: 0.019

```

[93]: # Guarda o modelo gerado
dic_melhores_modelos['split_perc_rf'] = []
dic_melhores_modelos['split_perc_rf'] = []

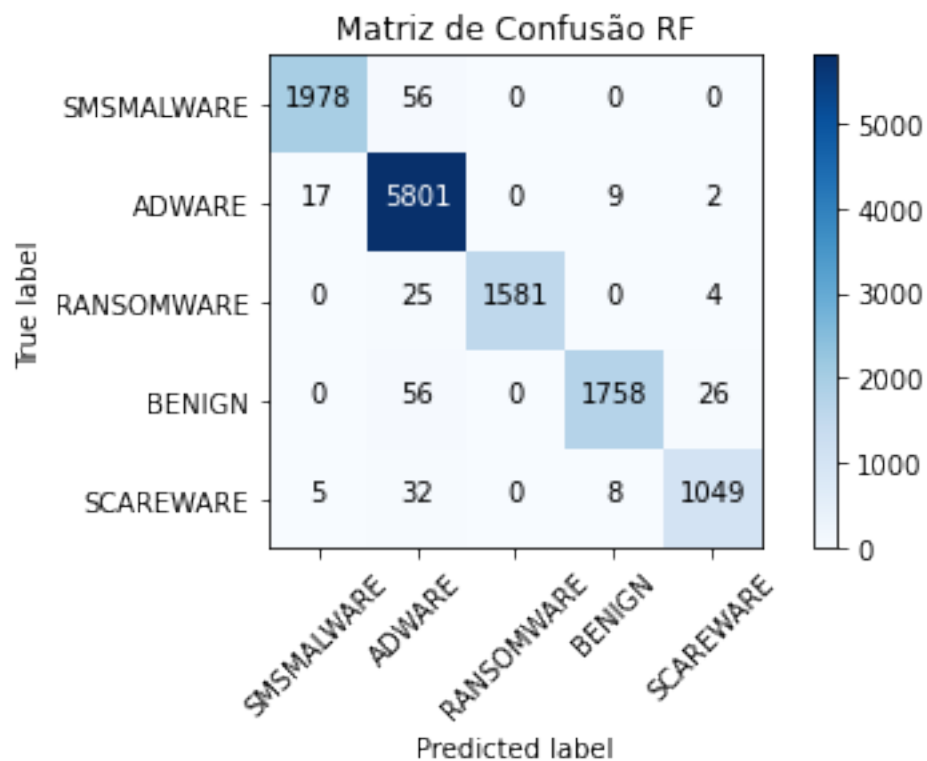
dic_melhores_modelos['split_perc_rf'].append(clf_rf)
dic_melhores_modelos['split_perc_rf'].append(accuracy)

```

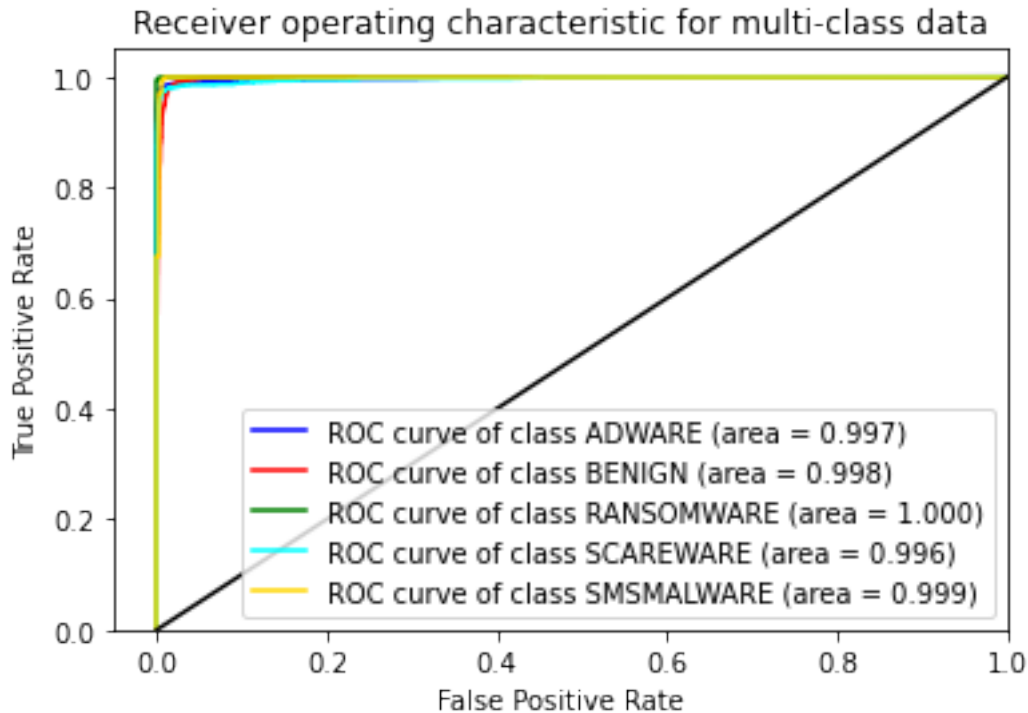
```

[94]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_teste_norm, y_pred,
                                     set(y_teste_norm),
                                     'Matriz de Confusão RF')

```



```
[127]: # Exibe curva ROC
funcoes_uteis.plot_roc_curve(clf_rf, X_train_norm, y_train_norm,
                             X_teste_norm, y_teste_norm)
```



3.3.4 MLP (*Multi-layer Perceptron*)

- Neste algoritmo serão variados o hiperparametro de número de camadas ocultas x número de neurônios, (*hidden_layer_sizes*) variando as camadas 1 até 2 e número de neurônios de 50 até 100 para detectar o melhor parâmetro.

```
[129]: tamanhos_camada_oculta = [(50,), (100,), (150,), (50, 50), (100, 100),
                                (150, 150)]
```

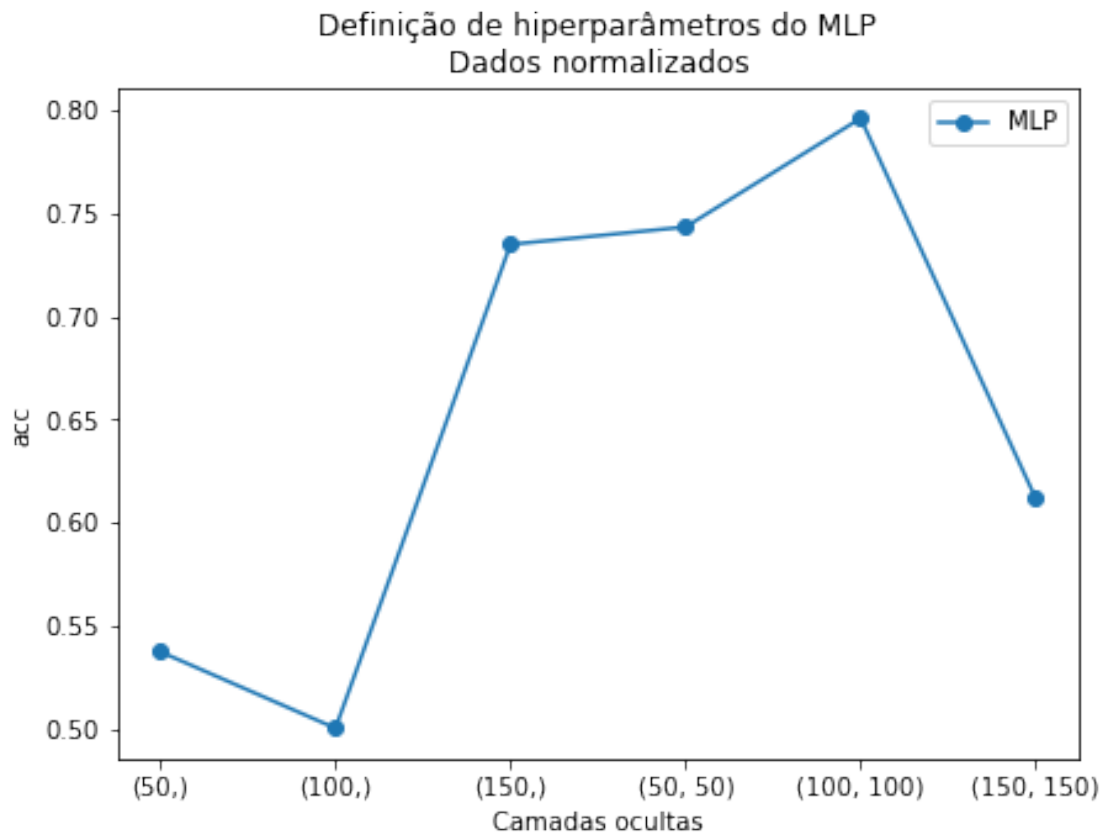
```
[144]: # Dados normalizados
acc = []
num_camadas = []
divisao = []
for camadas_ocultas in tamanhos_camada_oculta:
    # Cria modelo
    clf_mlp = MLPClassifier(hidden_layer_sizes=camadas_ocultas, random_state=1)
    # Treina modelo
    clf_mlp.fit(X_train_norm, y_train_norm)
    # Faz predição sobre os dados de teste
    y_pred = clf_mlp.predict(X_test_norm)
    # Calcula a acurácia
    acuracia = accuracy_score(y_test_norm, y_pred)
    acc.append(acuracia)
    num_camadas.append(str(camadas_ocultas))
```

```

divisao.append('MLP')

# Exibe os resultados
funcoes_uteis.plot_resultados(num_camadas, 'Camadas ocultas',
                              acc, 'acc', divisao,
                              'Definição de hiperparâmetros do MLP\nDados_
↳normalizados', figsize=(7,5))

```



```

[133]: # Dados não normalizados
acc = []
num_camadas = []
divisao = []
for camadas_ocultas in tamanhos_camada_oculta:
    # Cria modelo
    clf_mlp = MLPClassifier(hidden_layer_sizes=camadas_ocultas, random_state=1)
    # Treina modelo
    clf_mlp.fit(X_train, y_train)
    # Faz predição sobre os dados de teste
    y_pred = clf_mlp.predict(X_teste)
    # Calcula a acurácia

```

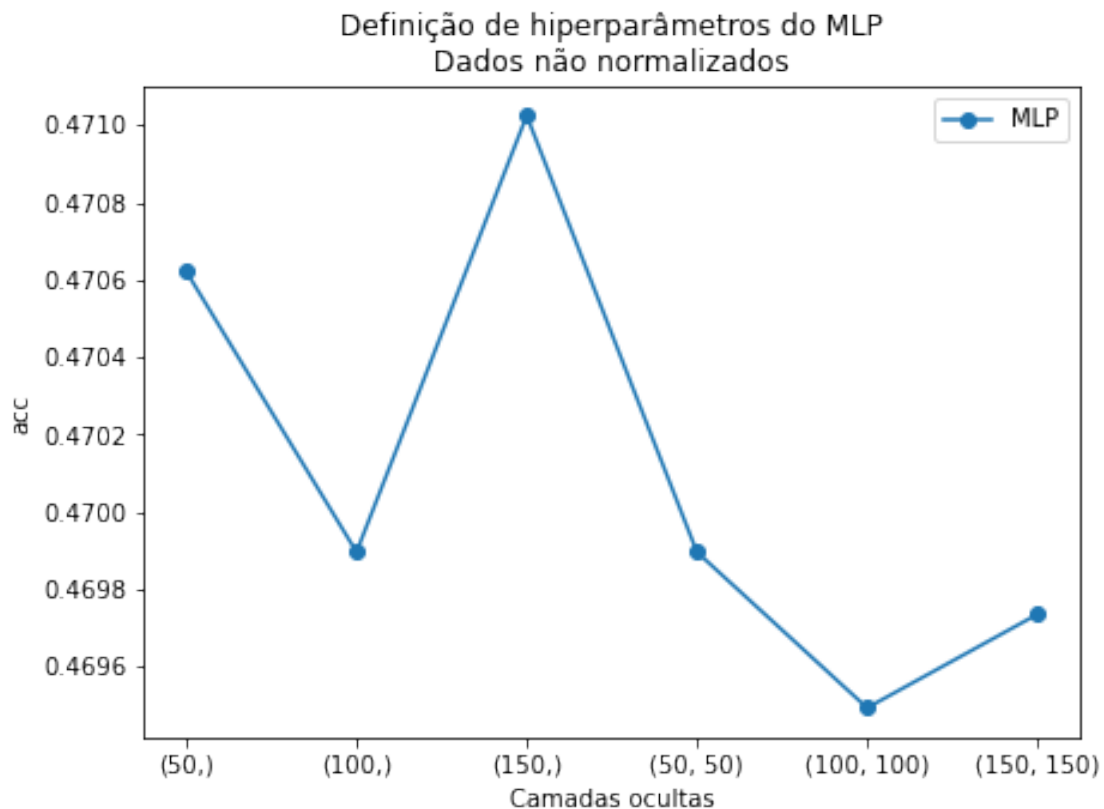


```

acuracia = accuracy_score(y_teste, y_pred)
acc.append(acuracia)
num_camadas.append(str(camadas_ocultas))
divisao.append('MLP')

# Exibe os resultados
funcoes_uteis.plot_resultados(num_camadas, 'Camadas ocultas',
                              acc, 'acc', divisao,
                              'Definição de hiperparâmetros do MLP\nDados não_
↪normalizados', figsize=(7,5))

```



Discussão

- Para o MLP a utilização dos **dados normalizados** chegaram a resultados melhores, pelos experimentos o melhor configuração das camadas ocultas é (100,100). Portanto o experimento para o *MLP* será executado com os seguintes hiperparâmetros e dados:
 - Dados: Normalizados
 - Configuração de camadas ocultas: (100,100)

```

[149]: # Cria modelo
clf_mlp = MLPClassifier(hidden_layer_sizes=(100,100), random_state=1)

```

```

# Treina modelo
clf_mlp.fit(X_train_norm, y_train_norm)
# Faz predição sobre os dados de teste
y_pred = clf_mlp.predict(X_teste_norm)
# Imprime resultados
print(classification_report(y_pred, y_teste, zero_division=True))

```

	precision	recall	f1-score	support
ADWARE	0.83	0.78	0.81	2144
BENIGN	0.90	0.80	0.85	6509
RANSOMWARE	0.83	0.92	0.87	1455
SCAREWARE	0.72	0.84	0.78	1586
SMSMALWARE	0.62	0.94	0.74	713
accuracy			0.83	12407
macro avg	0.78	0.86	0.81	12407
weighted avg	0.84	0.83	0.83	12407

```

[150]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_teste_norm, y_pred)
erro = 1 - accuracy
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))

```

Acurácia: 0.827

Erro: 0.173

```

[151]: # Guarda o modelo gerado
dic_melhores_modelos['split_perc_mlp'] = []
dic_melhores_modelos['split_perc_mlp'] = []

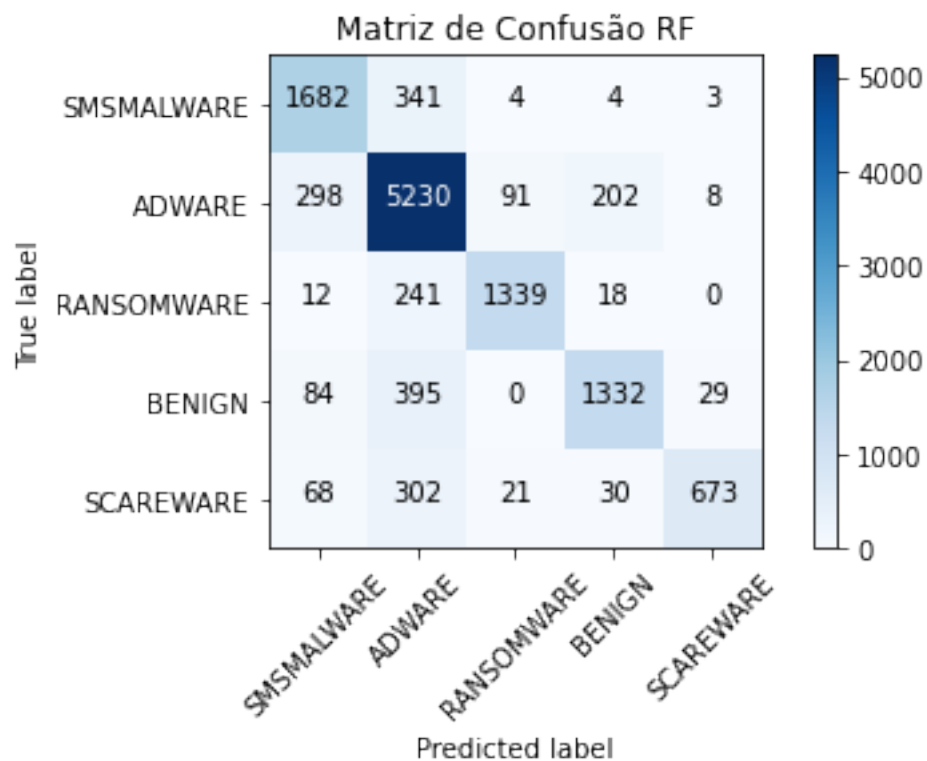
dic_melhores_modelos['split_perc_mlp'].append(clf_mlp)
dic_melhores_modelos['split_perc_mlp'].append(accuracy)

```

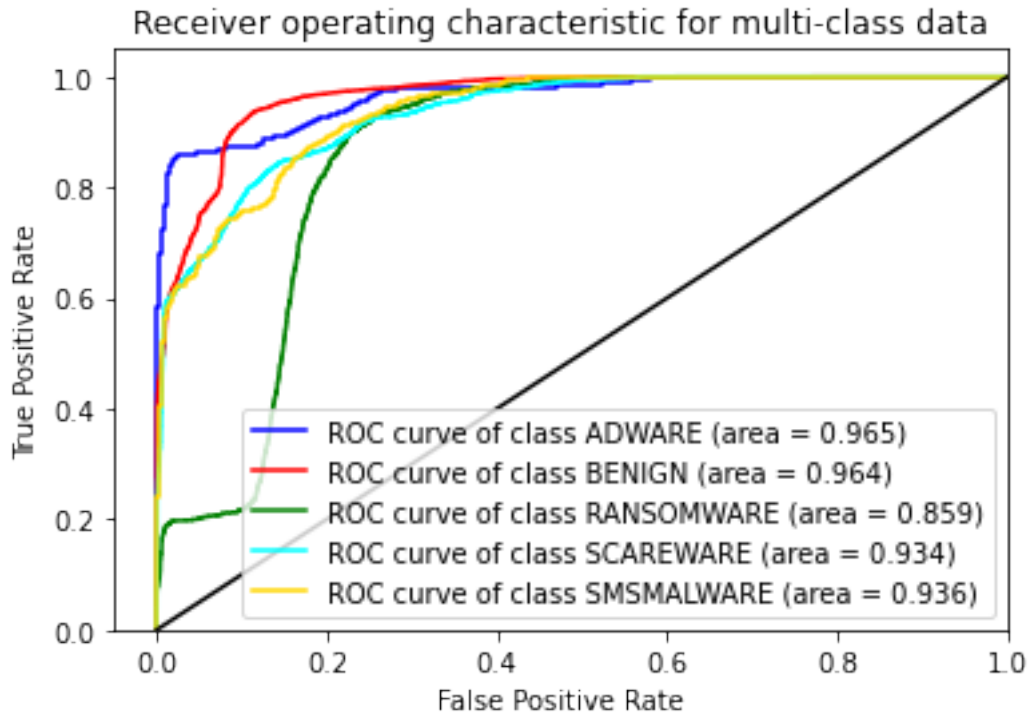
```

[152]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_teste_norm, y_pred,
                                     set(y_teste_norm),
                                     'Matriz de Confusão RF')

```



```
[153]: # Exibe curva ROC
funcoes_uteis.plot_roc_curve(clf_mlp, X_train_norm, y_train_norm,
                             X_teste_norm, y_teste_norm)
```

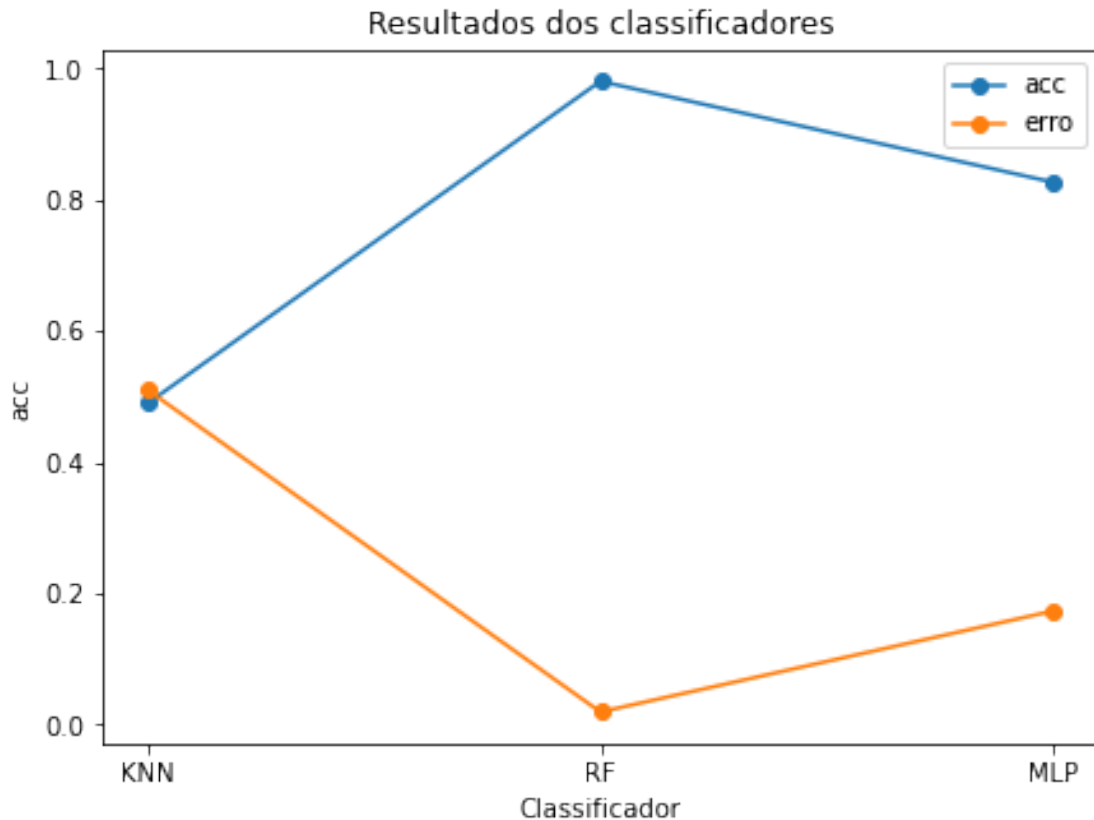


3.3.5 Comparação de Resultados e discussão

- É possível observar pelos experimentos anteriores e pelo gráfico abaixo que o KNN não consegue separar bem as classes. Isto pode ser observando tanto na matriz de confusão quanto pela curva ROC, uma vez que as linhas se aproximam muito do meio do gráfico. O *Random Forest* obteve os melhores resultados para predição de todas as classes, com índice muito baixo de falsos positivos, conforme pode ser visto pela curva ROC do algoritmo. O MLP consegue prever razoavelmente bem algumas classes, entretanto tem dificuldades para prever SMSMALWARE e SCAREWARE.
- Entre todos os modelos testados até este ponto, o *Random Forest* é o melhor modelo.

```
[162]: classificador_split_test = ['KNN', 'RF', 'MLP', 'KNN', 'RF', 'MLP',]
acuracias_split_test = [0.490, 0.981, 0.827, 1-0.490, 1-0.981, 1-0.827]
divisao_split_test = ['acc', 'acc', 'acc', 'erro', 'erro', 'erro']

# Exibe os resultados
funcoes_uteis.plot_resultados(classificador_split_test, 'Classificador',
                              acuracias_split_test, 'acc', divisao_split_test,
                              'Resultados dos classificadores', figsize=(7,5))
```



3.4 Treinamento e teste com validação cruzada

- Nesta etapa serão realizados treinamentos e teste com validação cruzada em 5 partes (*folds*). Para o treinamento e teste serão utilizados os 80% dos dados separados anteriormente.
- Os treinamentos serão realizados considerando os melhores hiperparâmetros encontrados anteriormente para os algoritmos. Para cada etapa do treinamento será exibido a matriz de confusão, Curva ROC, acurácia e erro do modelo.

3.4.1 Preparação dos dados para treinamento, normalizados e não normalizados

```
[19]: # Remove a coluna NOME_APP
df_treinamento_norm = df_treinamento_norm.drop('NOME_APP', axis=1)
df_treinamento = df_treinamento.drop('NOME_APP', axis=1)
```

```
[20]: # Dados totais de treinamento
X_treinamento_norm = df_treinamento_norm.iloc[:,0:-1].values
y_treinamento_norm = df_treinamento_norm.iloc[:, -1].values

X_treinamento = df_treinamento.iloc[:,0:-1].values
y_treinamento = df_treinamento.iloc[:, -1].values
```

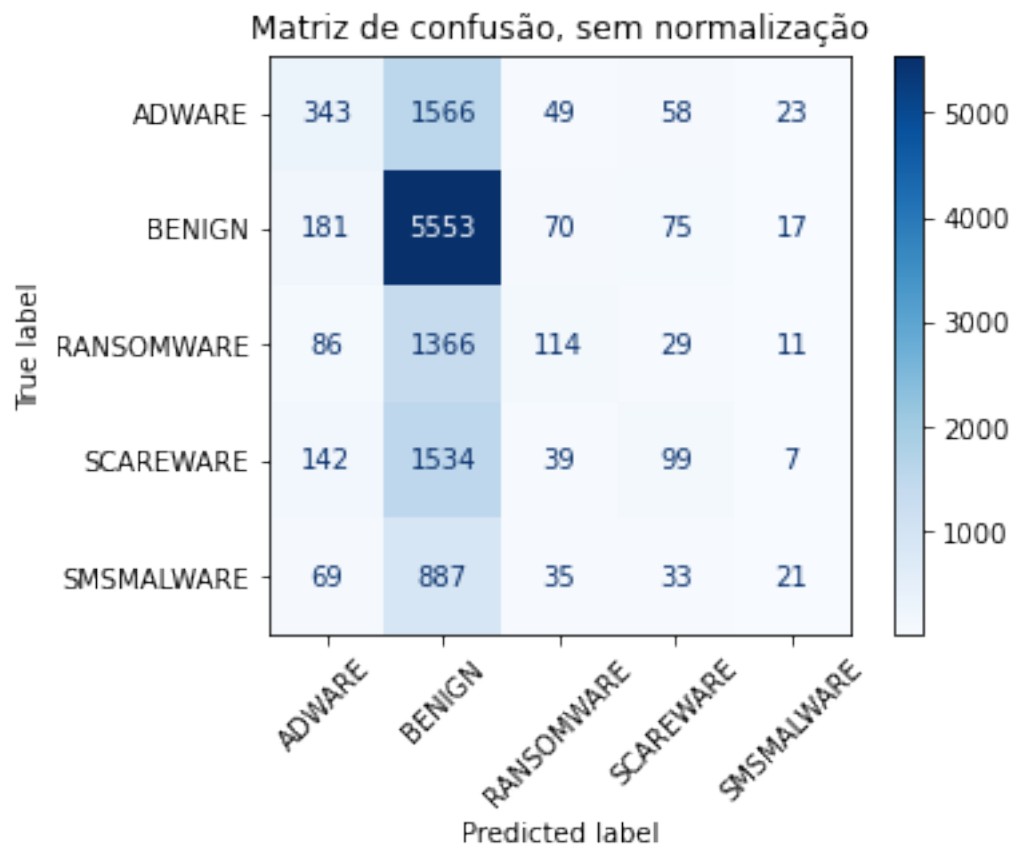
3.4.2 KNN (*K-nearest neighbors algorithm*)

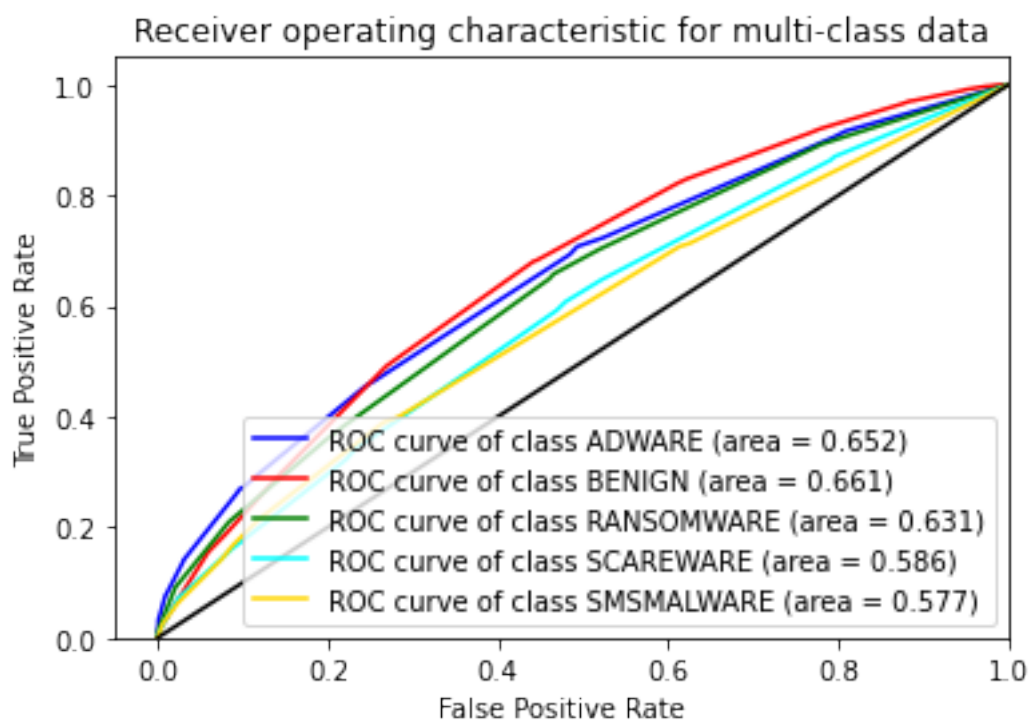
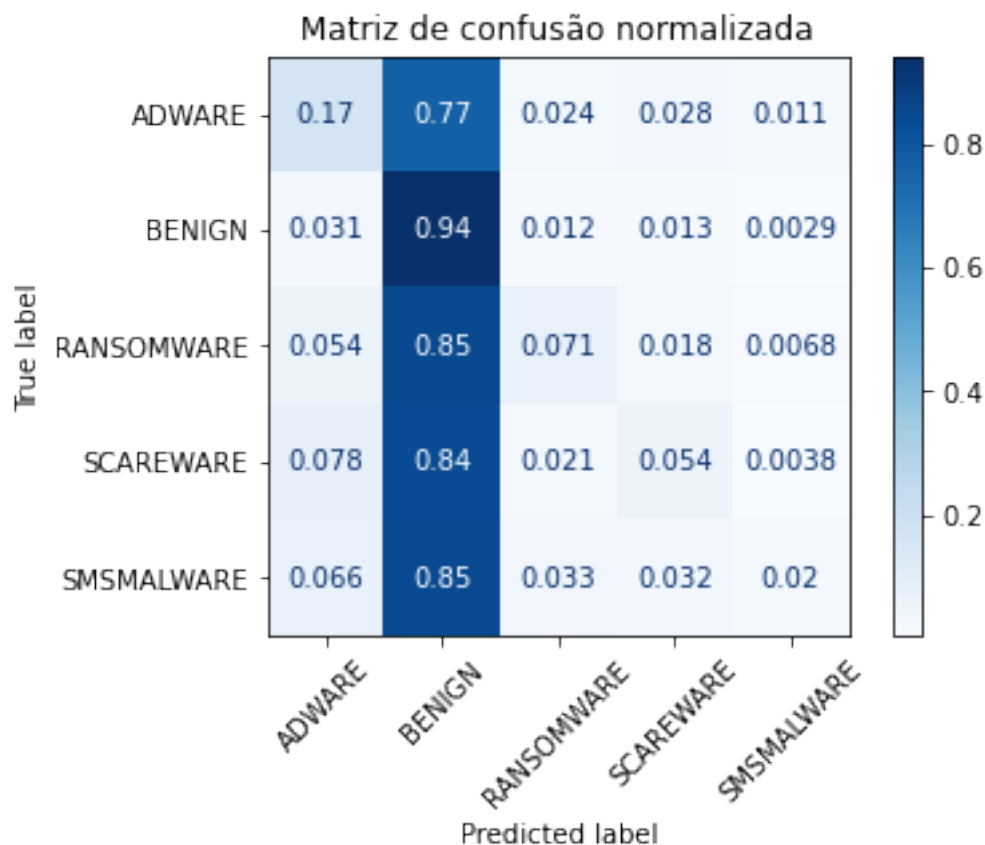
- Neste algoritmo utilizados serão os mesmos hiperparâmetros da etapa *percentage split*. Portanto o experimento para o KNN será executado com os seguintes hiperparâmetros e dados:
 - Dados: Normalizados
 - Número de vizinhos (k): 13
 - Métrica de distância: *manhattan*

```
[21]: clf_kNN = KNeighborsClassifier(n_neighbors=13, metric='manhattan')  
  
scores, models, folds = funcoes_uteis.k_fold_train(clf_kNN,  
↳df_treinamento_norm, clf_name='KNN')
```

Acurácia: 0.494

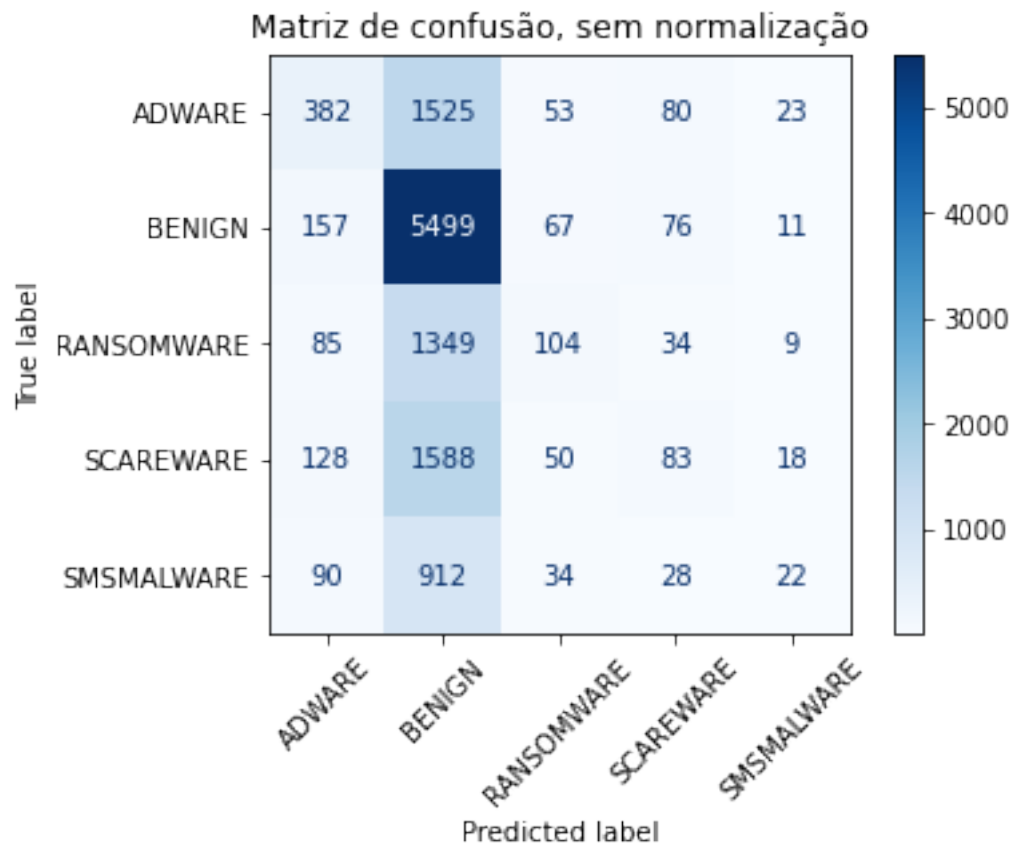
Erro: 0.506

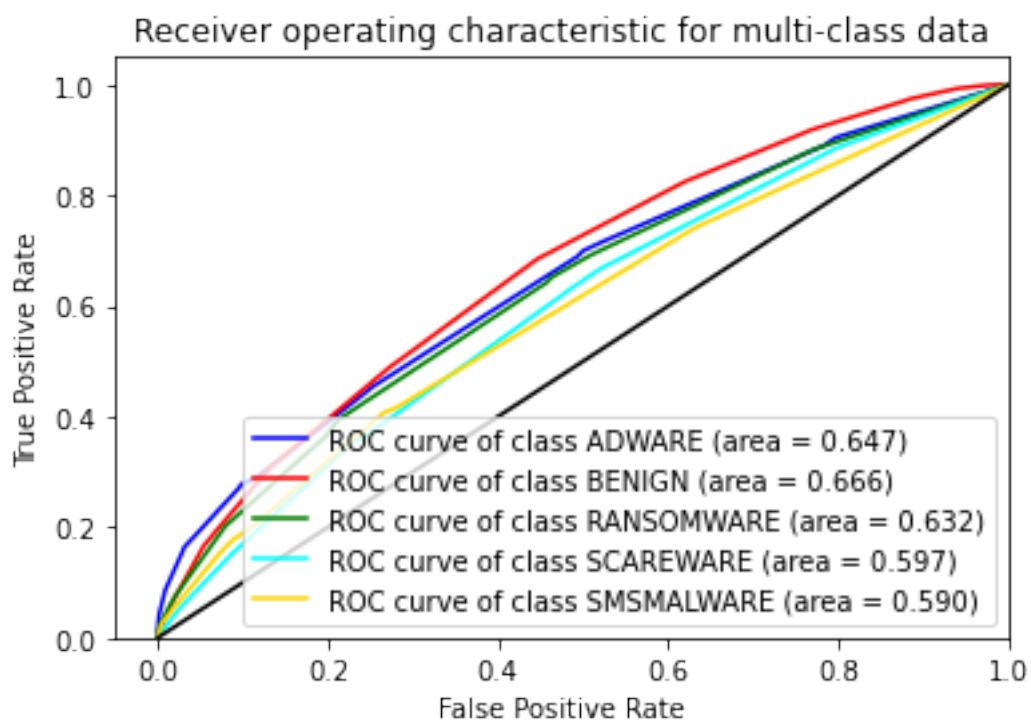
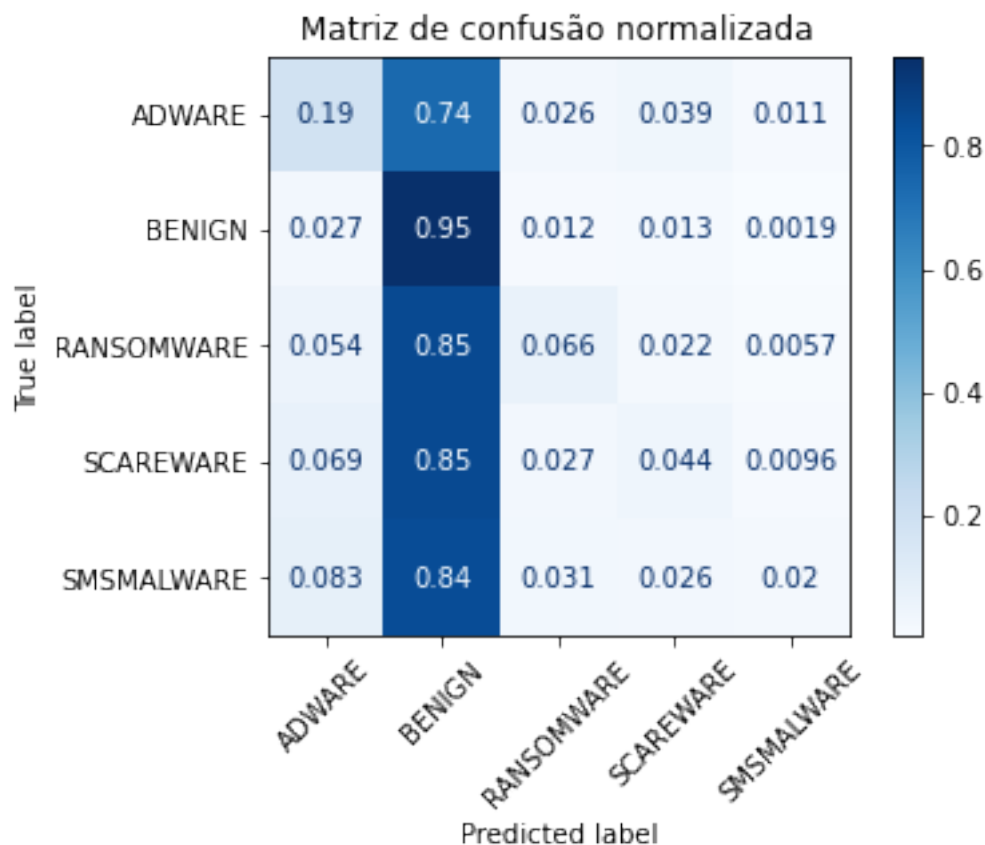




Acurácia: 0.491

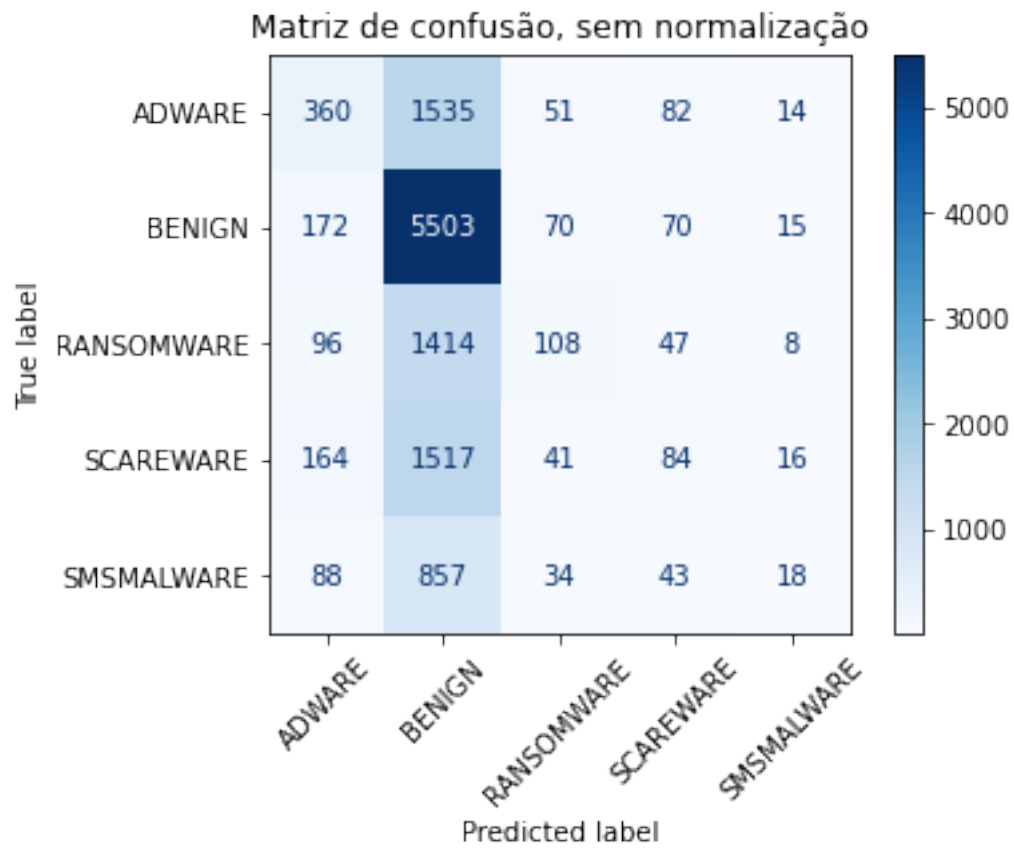
Erro: 0.509

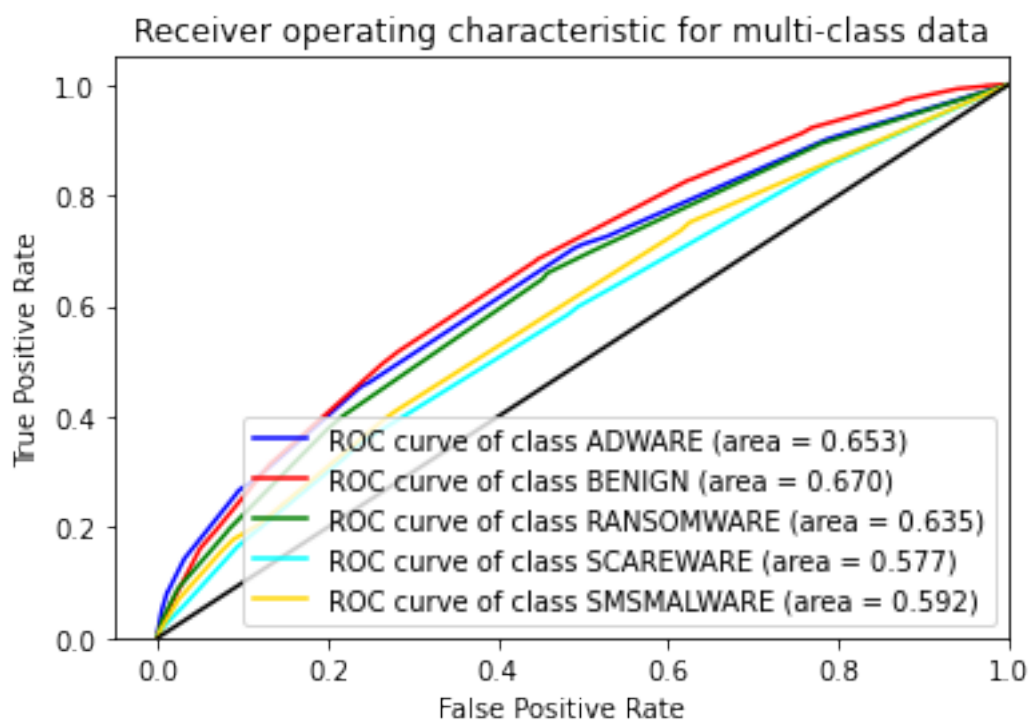
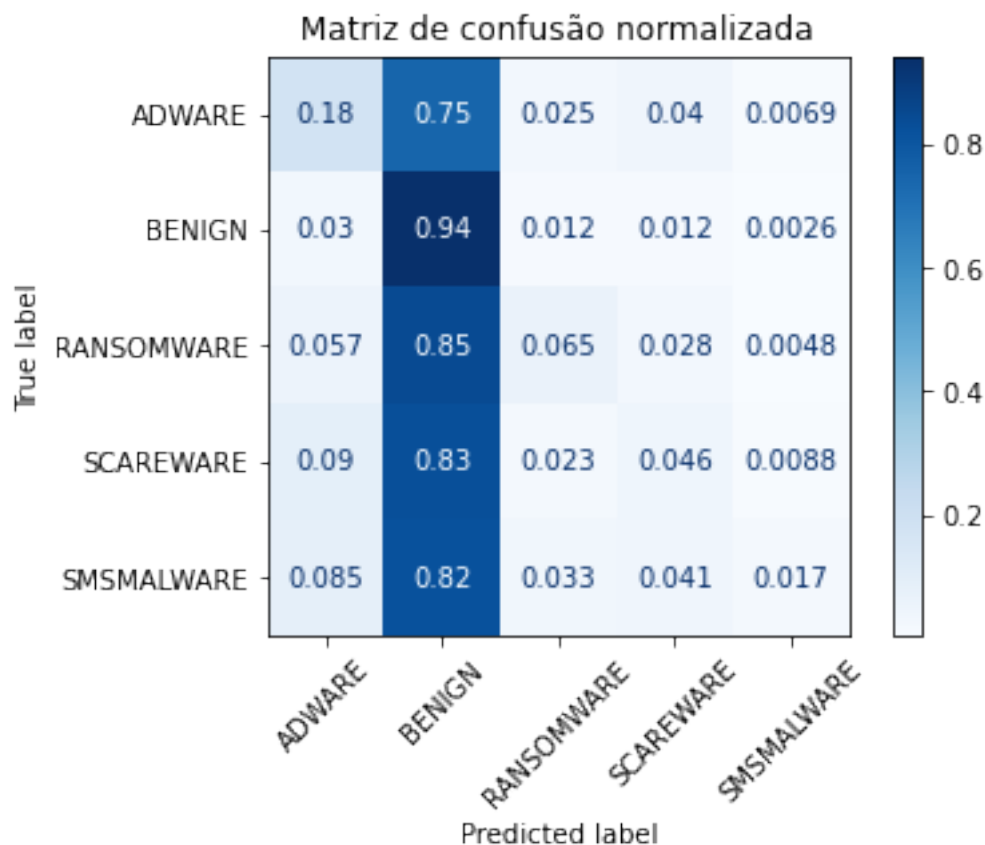




Acurácia: 0.489

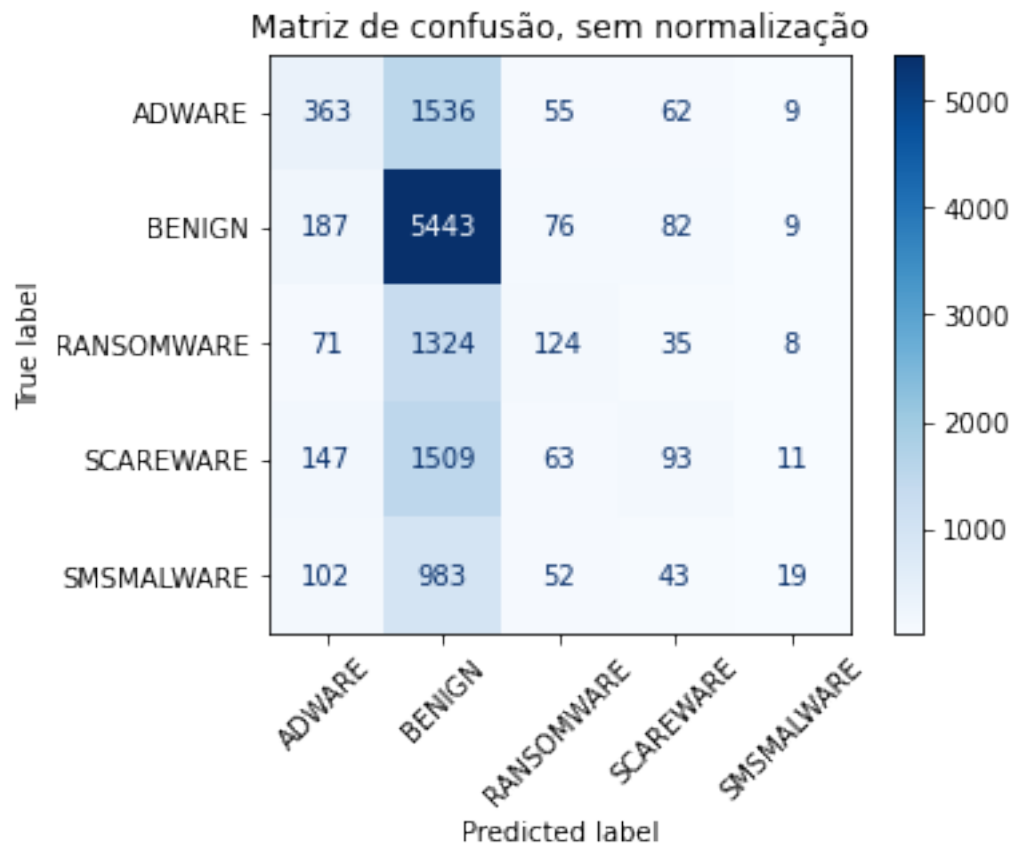
Erro: 0.511

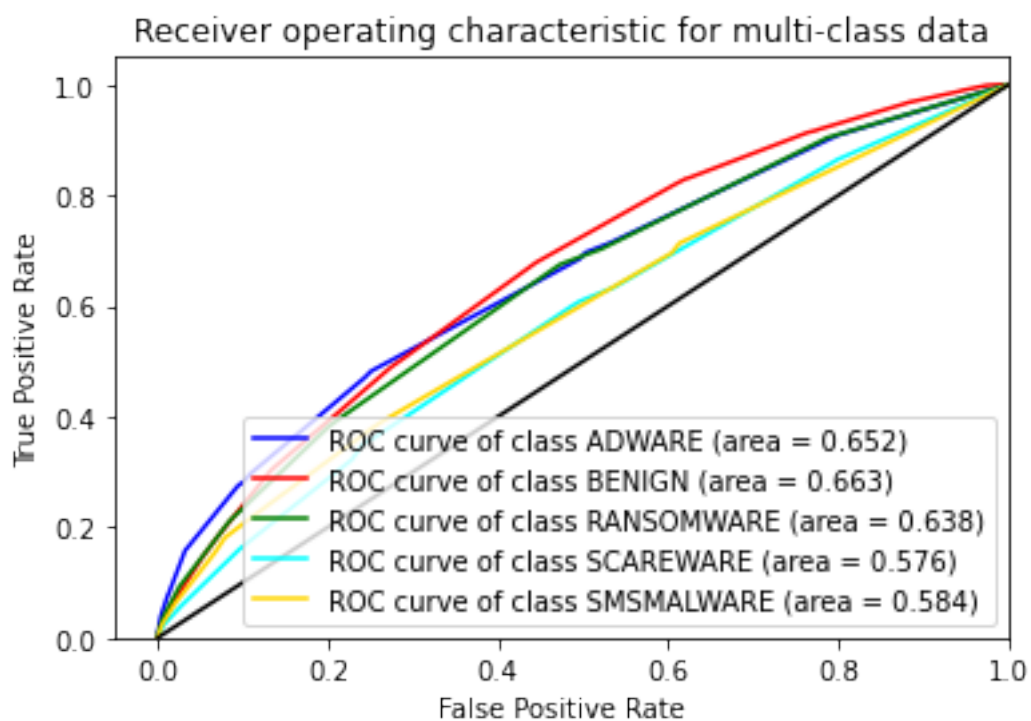
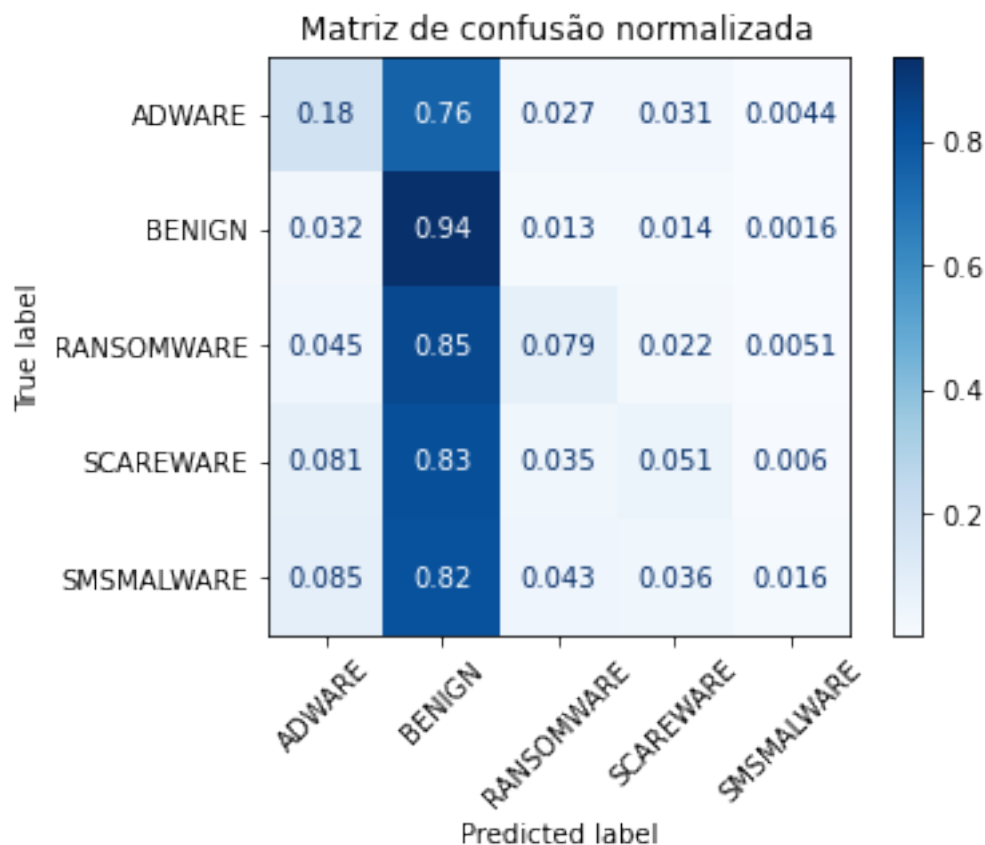




Acurácia: 0.487

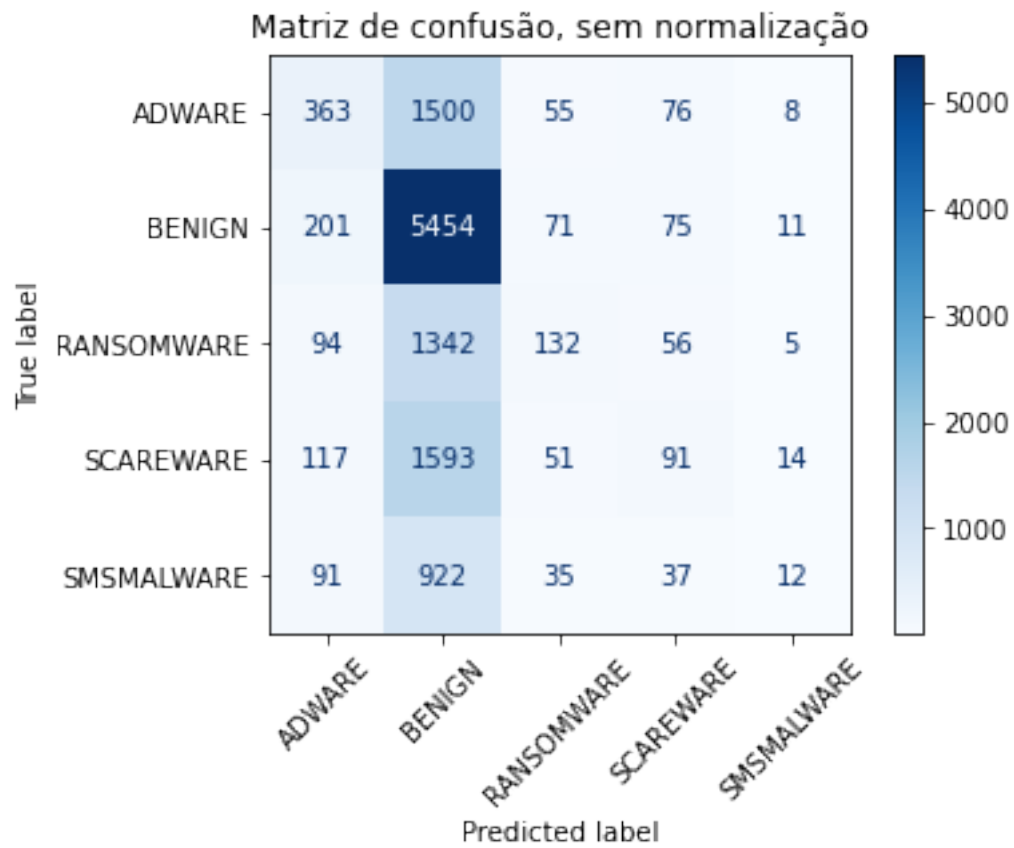
Erro: 0.513

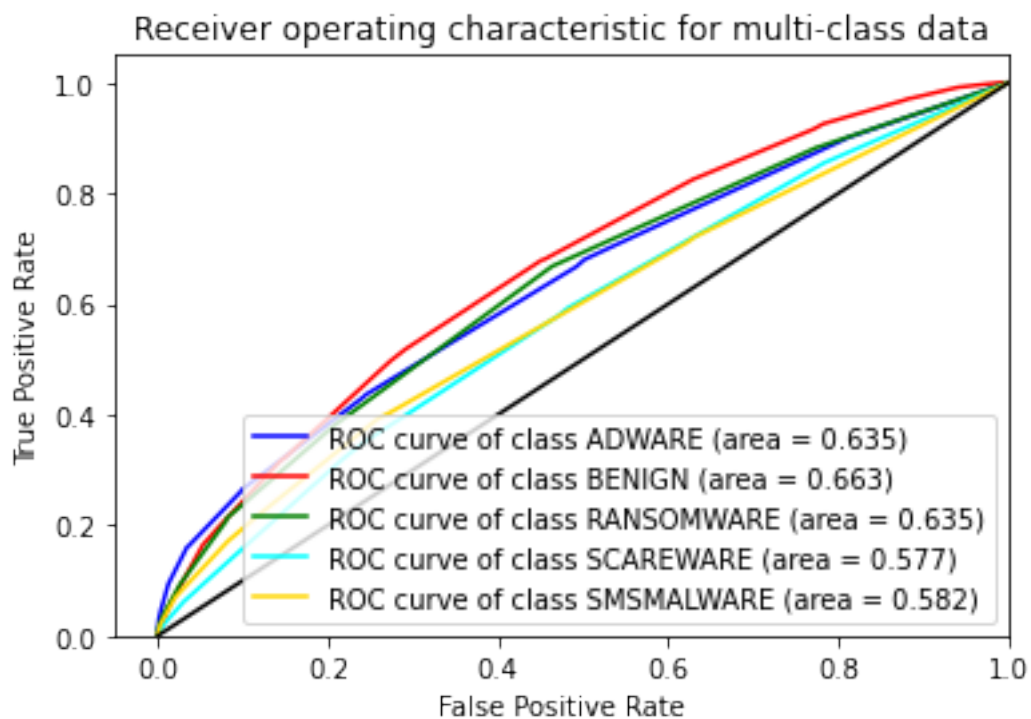
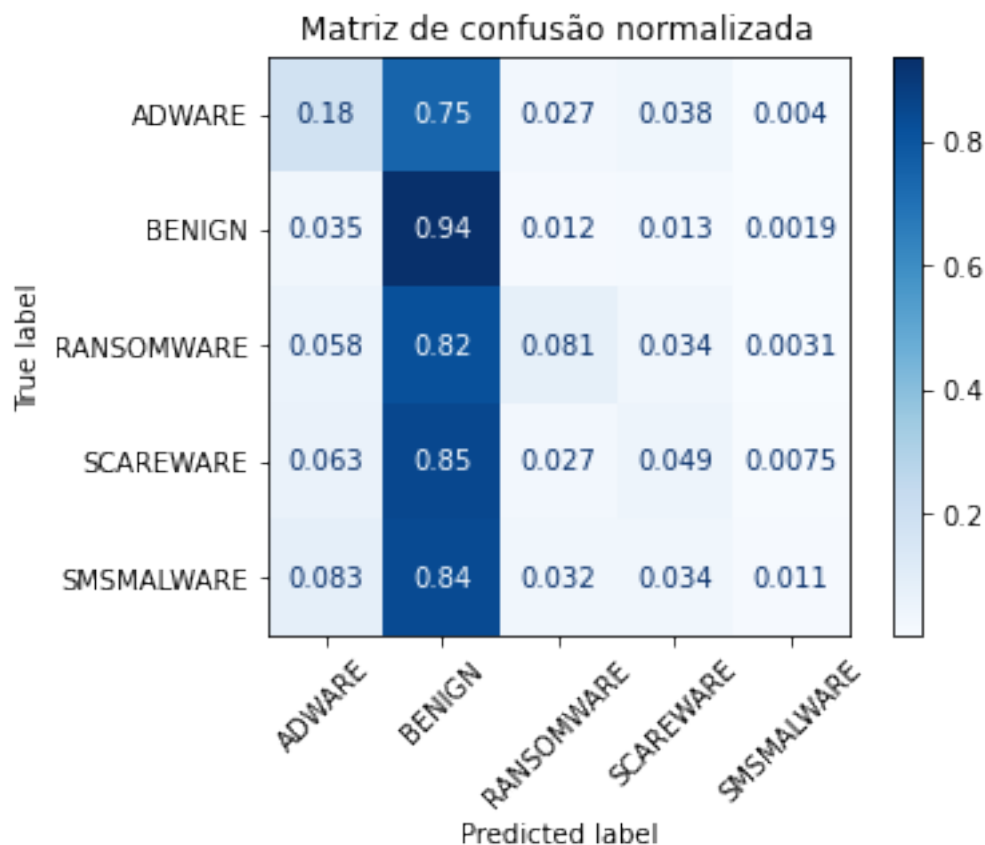




Acurácia: 0.488

Erro: 0.512





Média: 0.48985209721735484

```
[23]: media_acc = sum(scores)/len(scores)
media_erro = 1 - media_acc
print("Acurácia média:", media_acc)
print("Erro médio:", media_erro)
```

Acurácia média: 0.48985209721735484

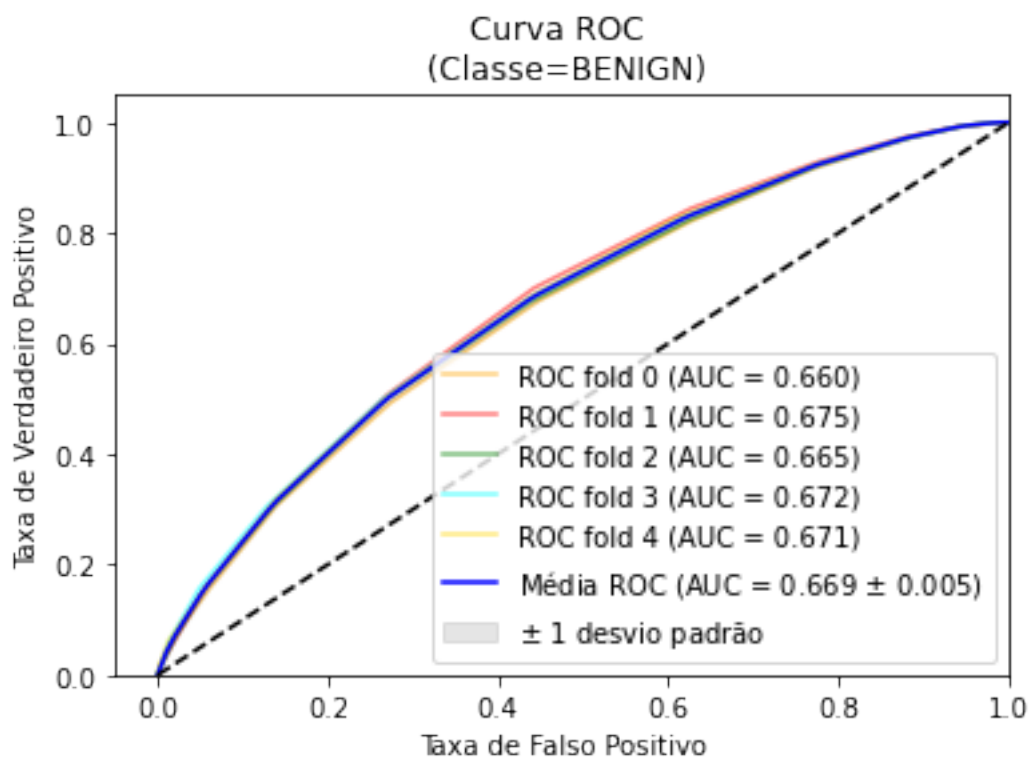
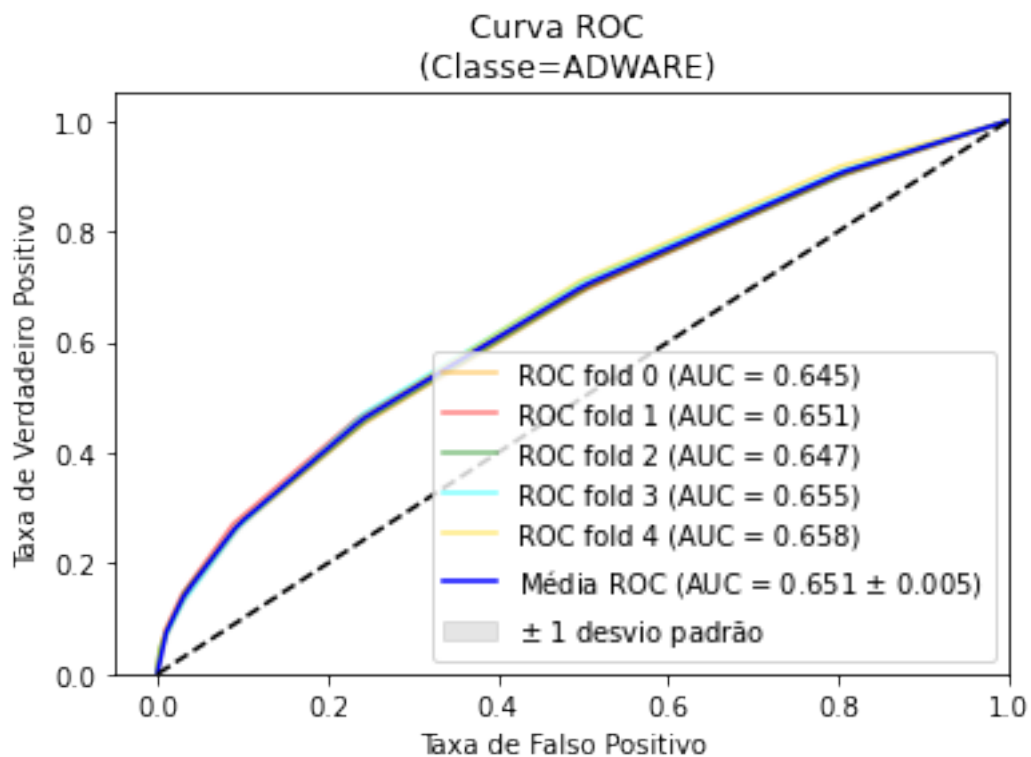
Erro médio: 0.5101479027826452

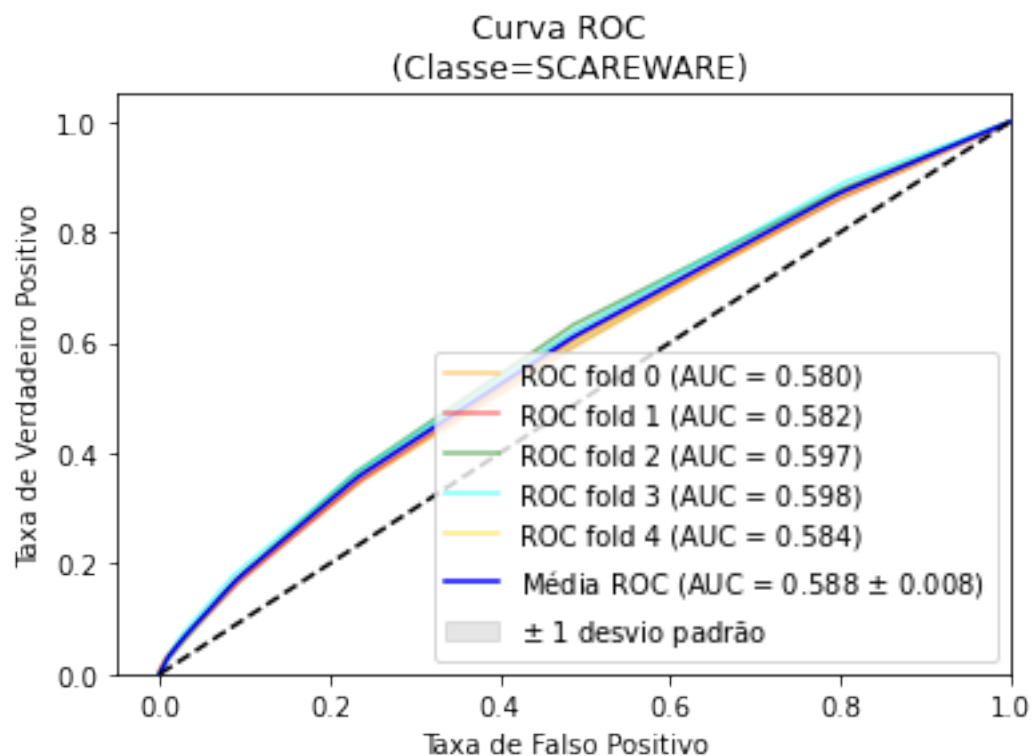
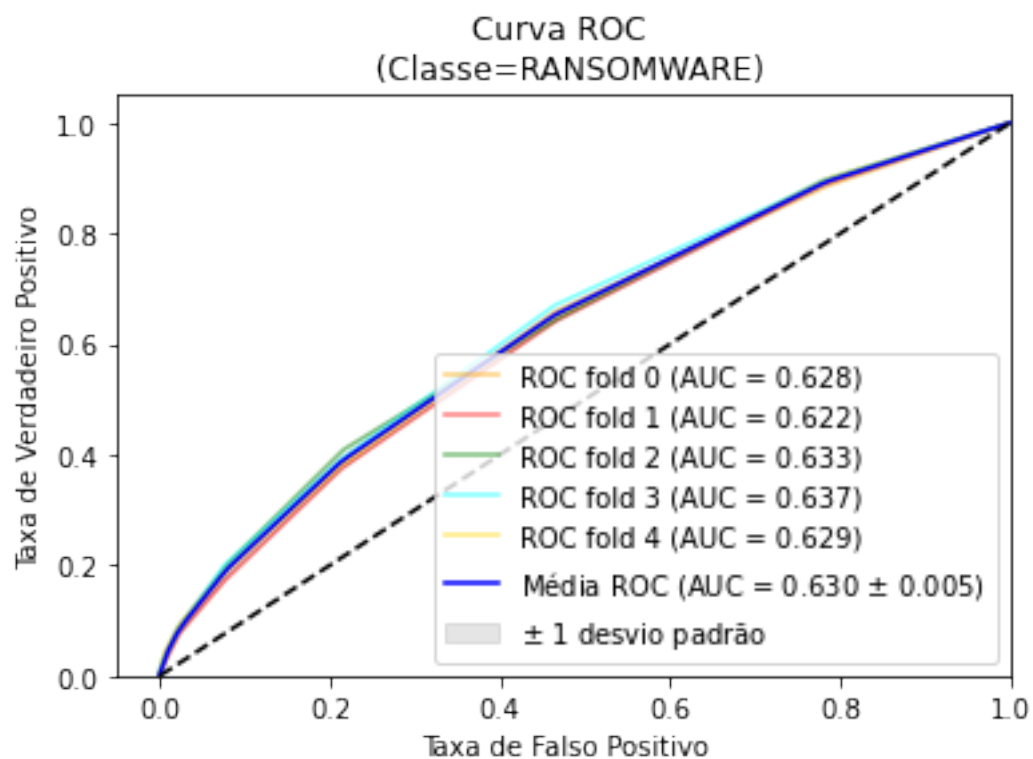
```
[33]: scores_array = np.array(scores)
models_array = np.array(models)
folds_array = np.array(folds)
index = np.where(scores_array == max(scores_array))
melhor_model_KNN_k_fold = models_array[index[0]][0]
KNN_k_fold = folds_array[index[0]]
```

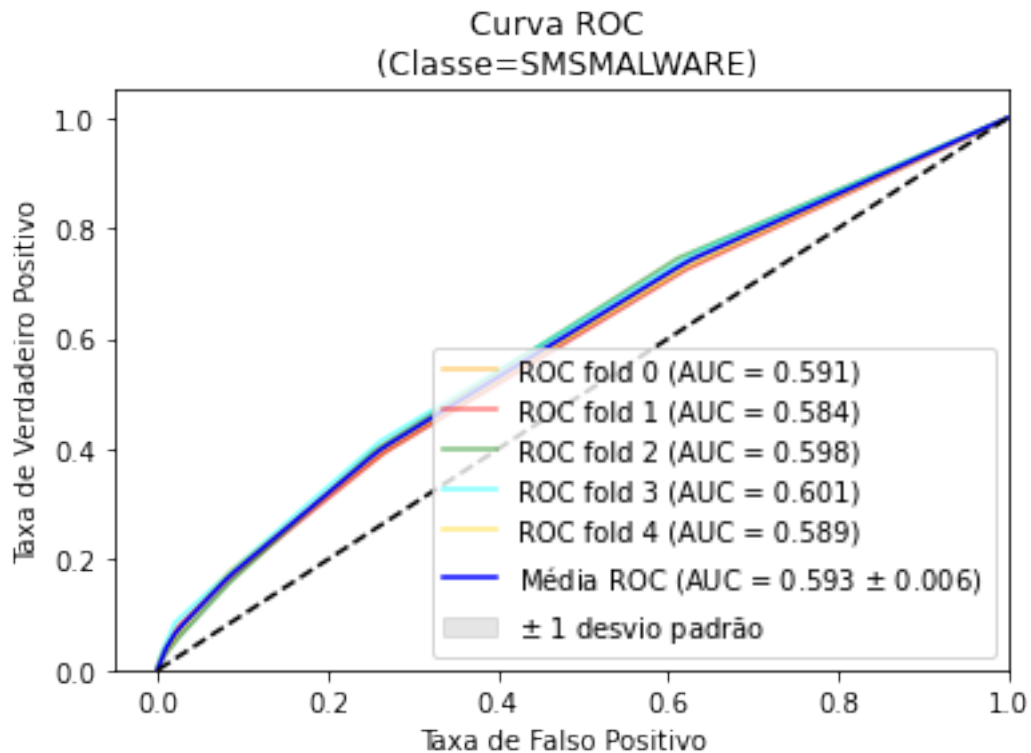
```
[37]: # Guarda o modelo gerado
dic_melhores_modelos['melhor_val_cruz_knn'] = []
dic_melhores_modelos['melhor_val_cruz_knn'] = []
dic_melhores_modelos['melhor_val_cruz_knn'] = []

dic_melhores_modelos['melhor_val_cruz_knn'].append(melhor_model_KNN_k_fold)
dic_melhores_modelos['melhor_val_cruz_knn'].append(media_acc)
dic_melhores_modelos['melhor_val_cruz_knn'].append(KNN_k_fold)
```

```
[28]: # Curva ROC por classe
funcoes_uteis.plot_roc_k_fold(clf_kNN, X_treinamento_norm, y_treinamento_norm, 5)
```





3.4.3 *Random Forest*

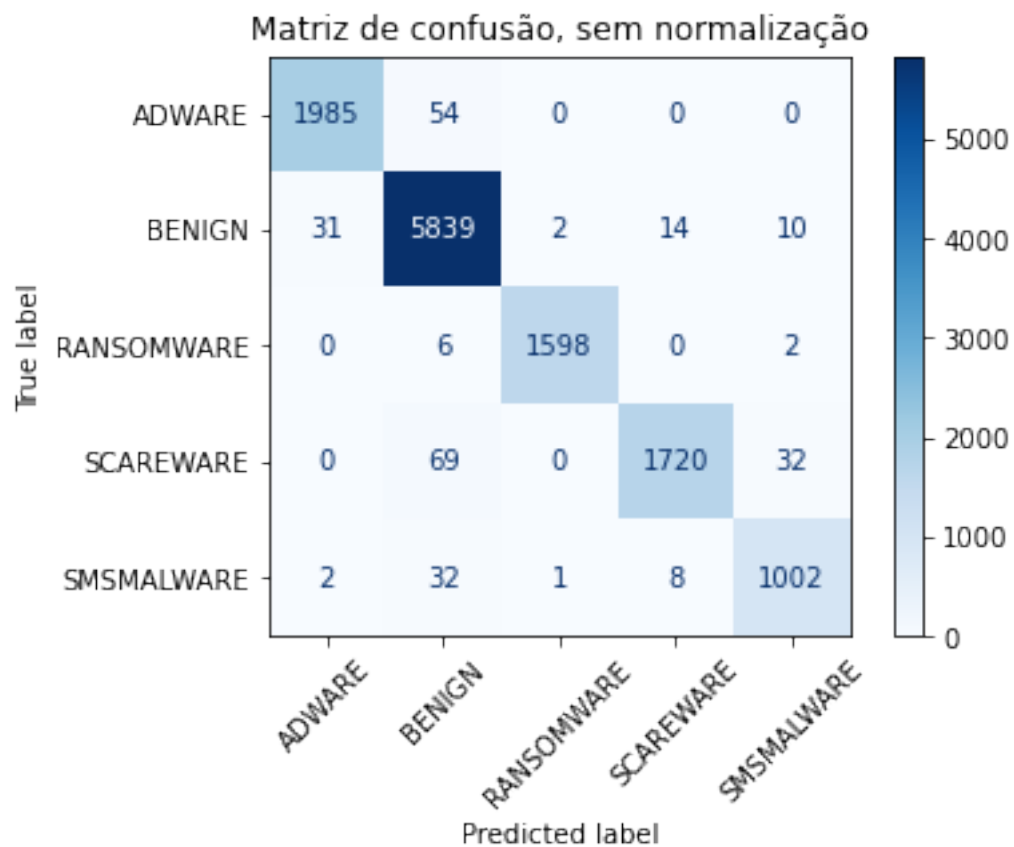
- Neste algoritmo foi utilizados os mesmos hiperparametros da etapa *percentage split*. Portanto o experimento para o *Random Forest* será executado com o seguinte hiperparâmetros e dados:
 - Dados: Normalizados
 - Número de estimadores: 50

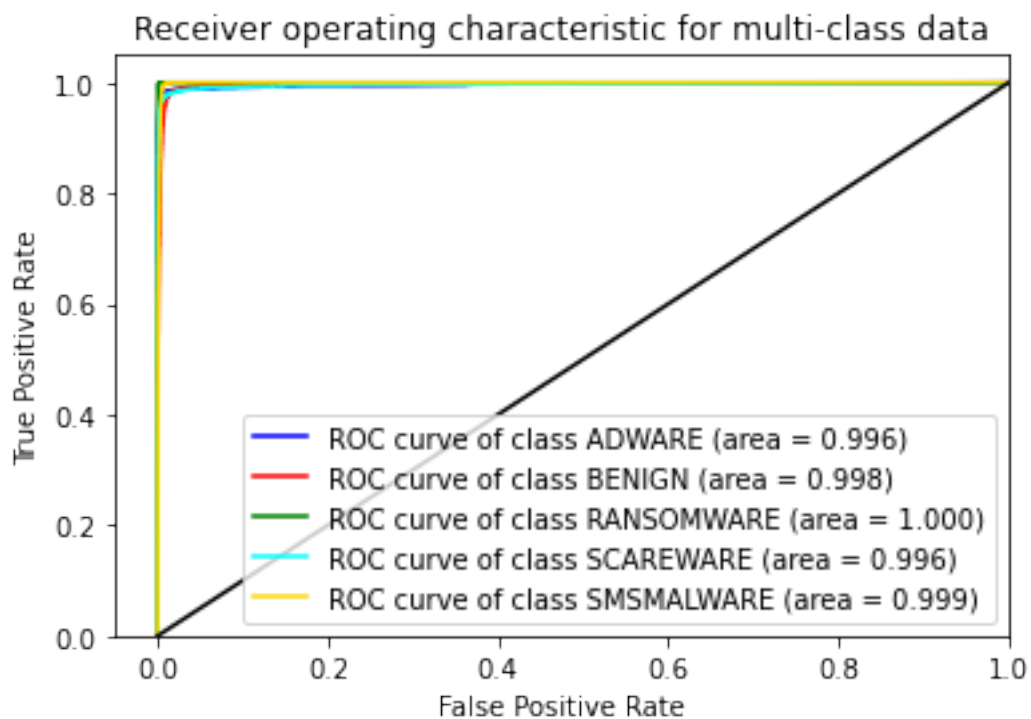
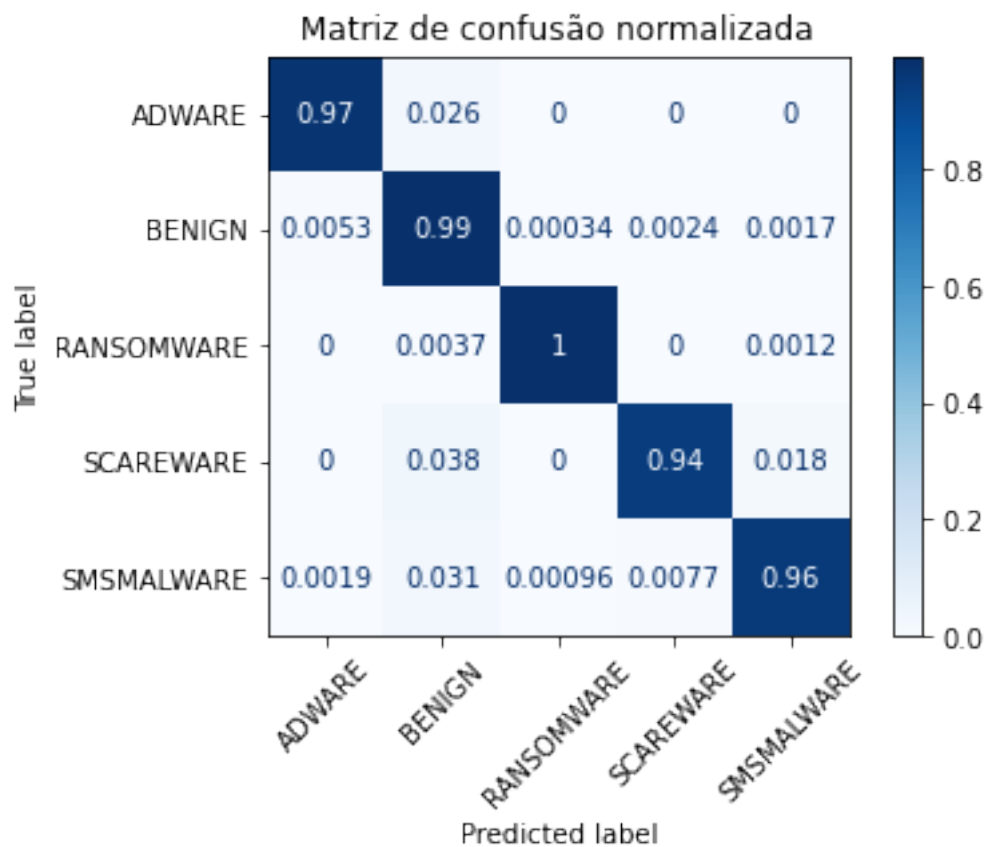
```
[73]: clf_rf = RandomForestClassifier(n_estimators = 50, random_state=0)

scores, models, folds = funcoes_uteis.k_fold_train(clf_rf, df_treinamento_norm,
↳ clf_name='RF')
```

Acurácia: 0.979

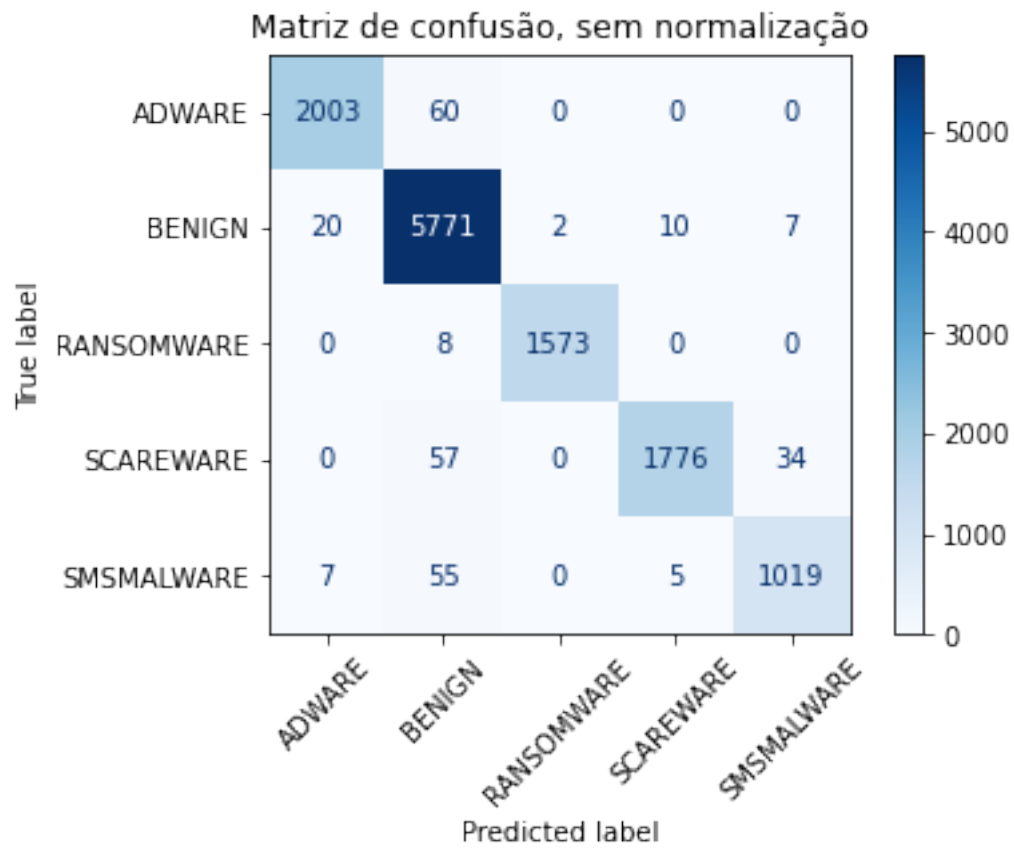
Erro: 0.021

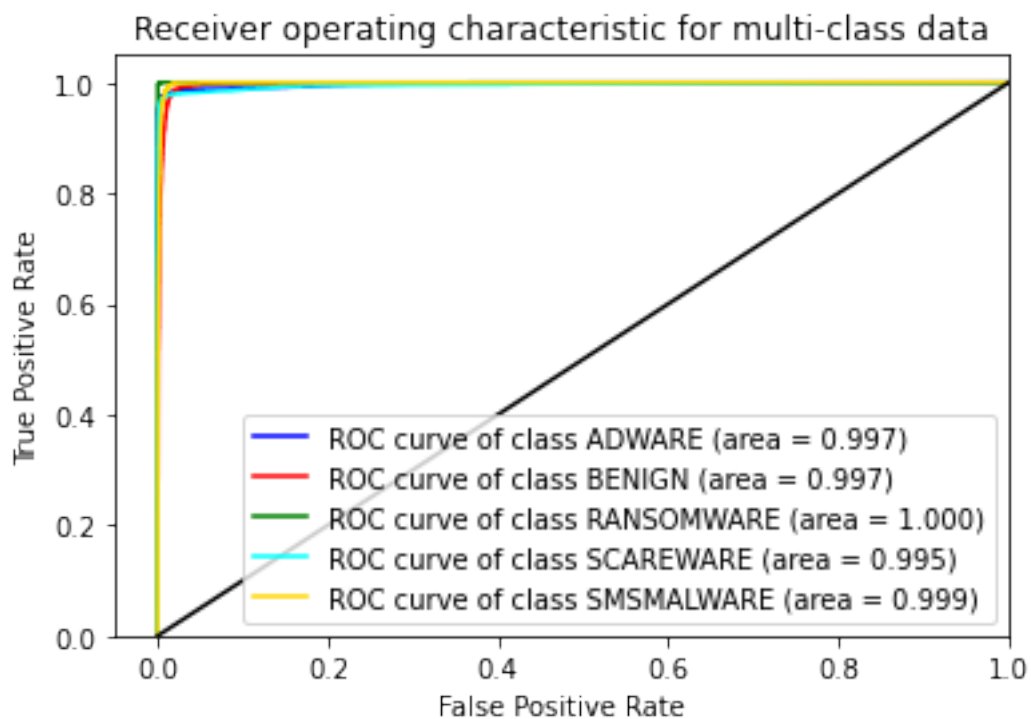
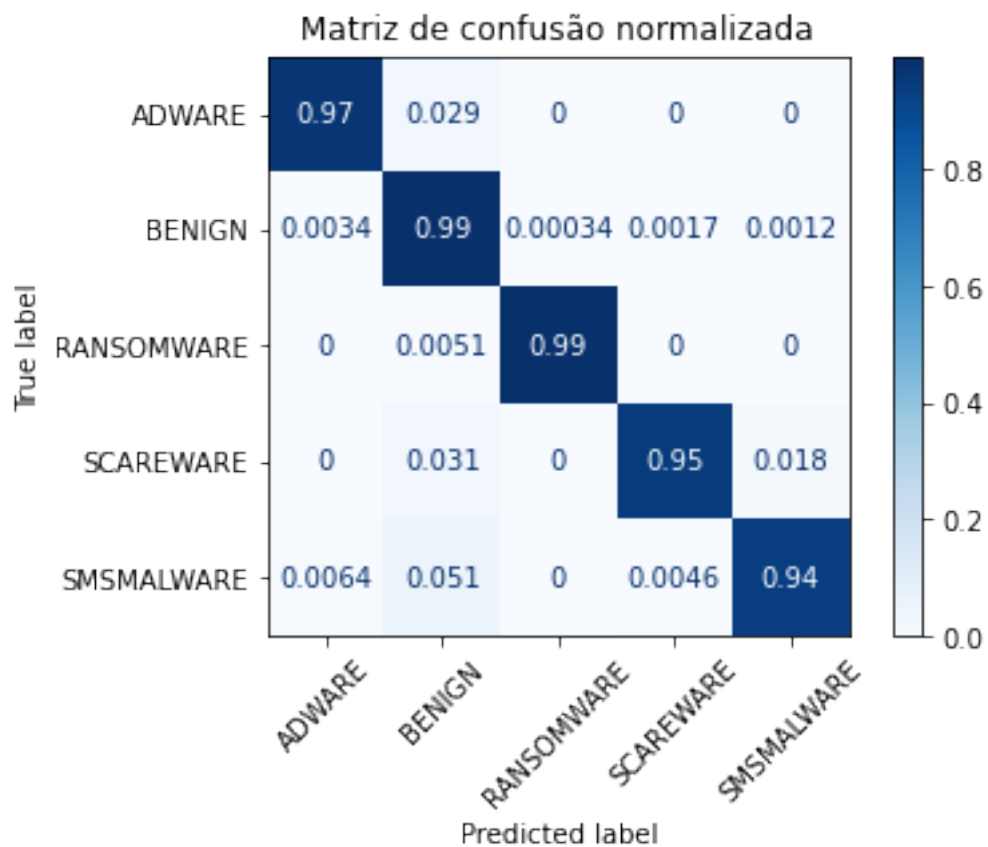




Acurácia: 0.979

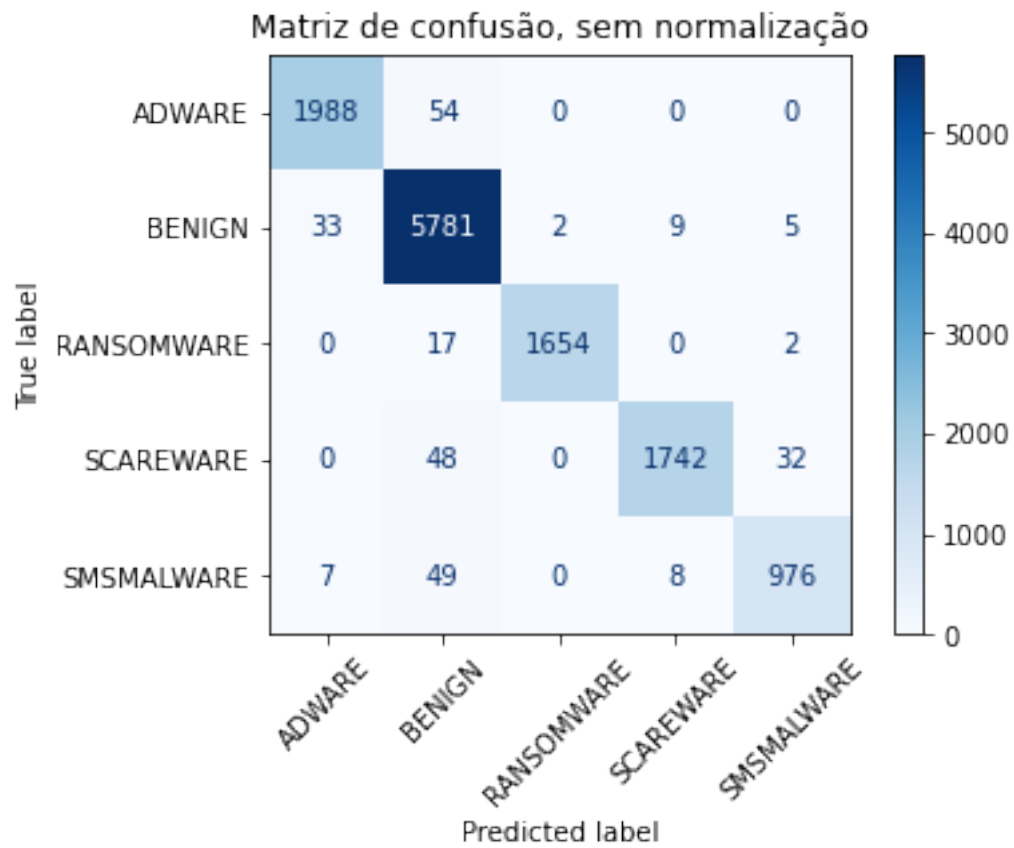
Erro: 0.021

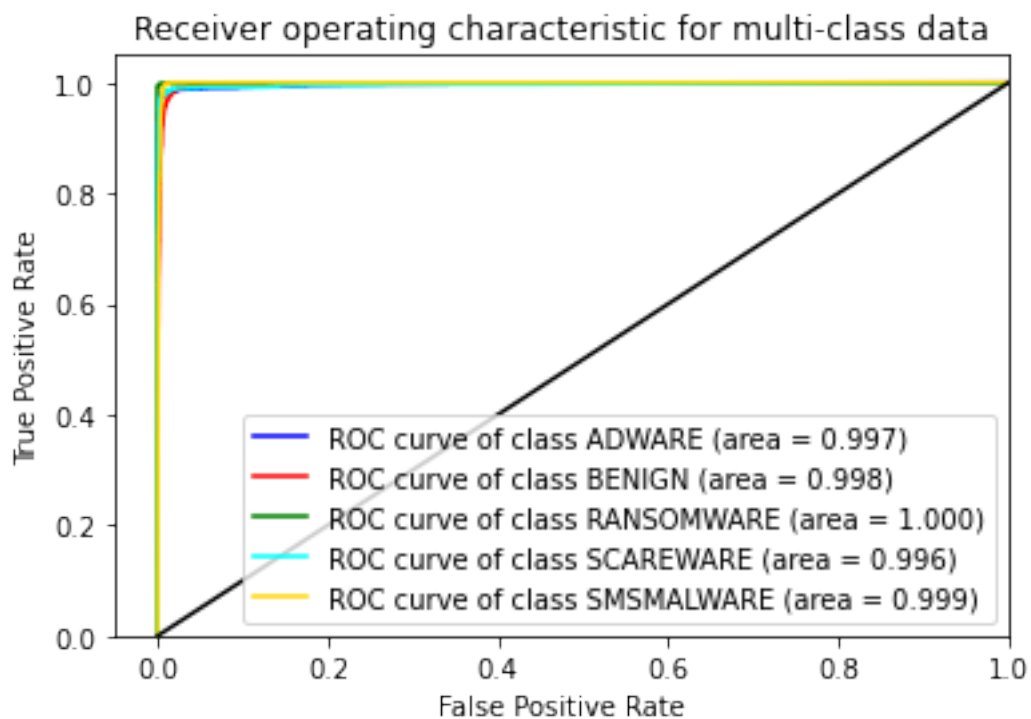
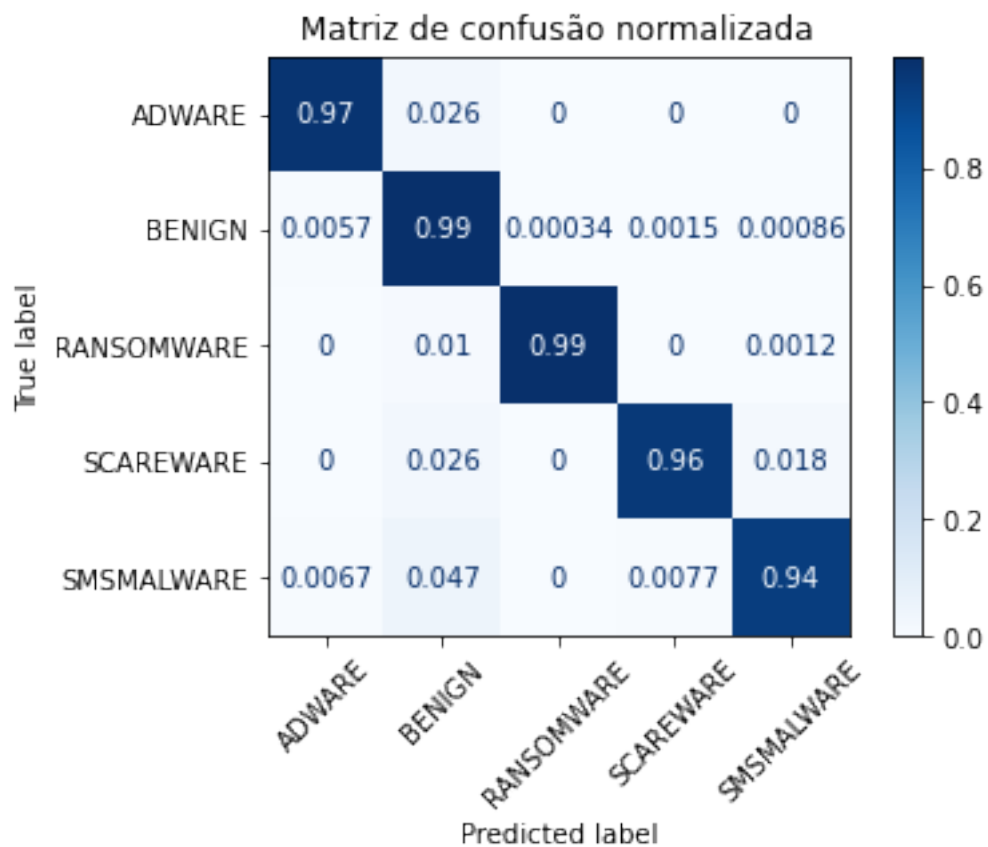




Acurácia: 0.979

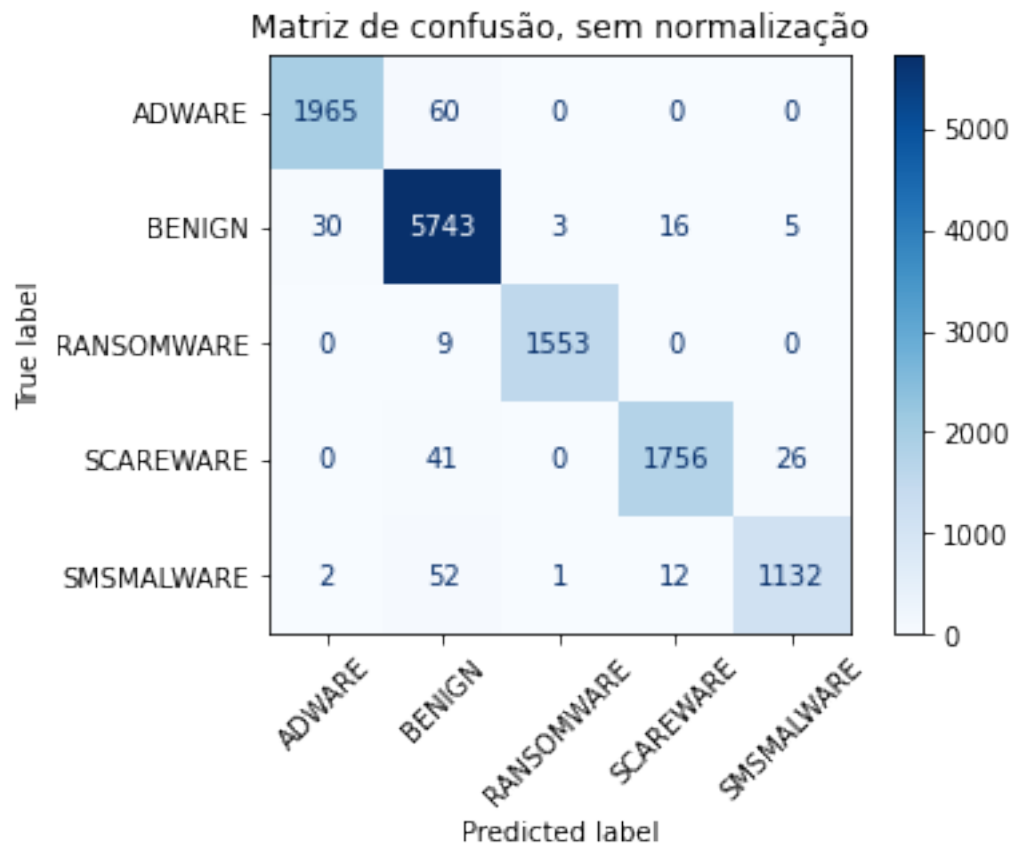
Erro: 0.021

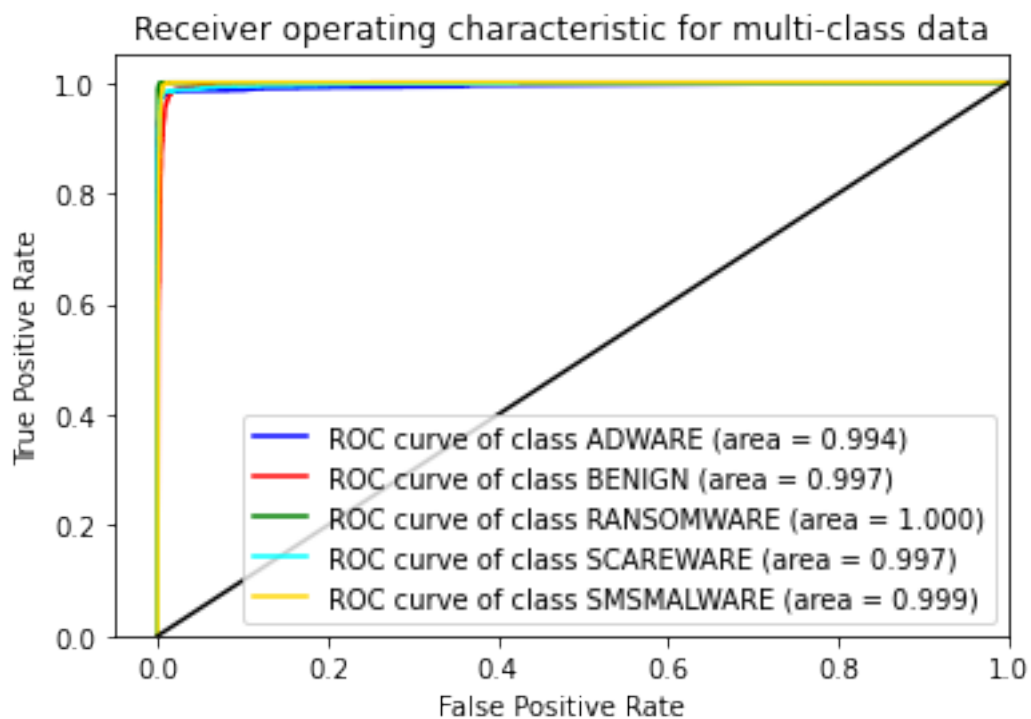
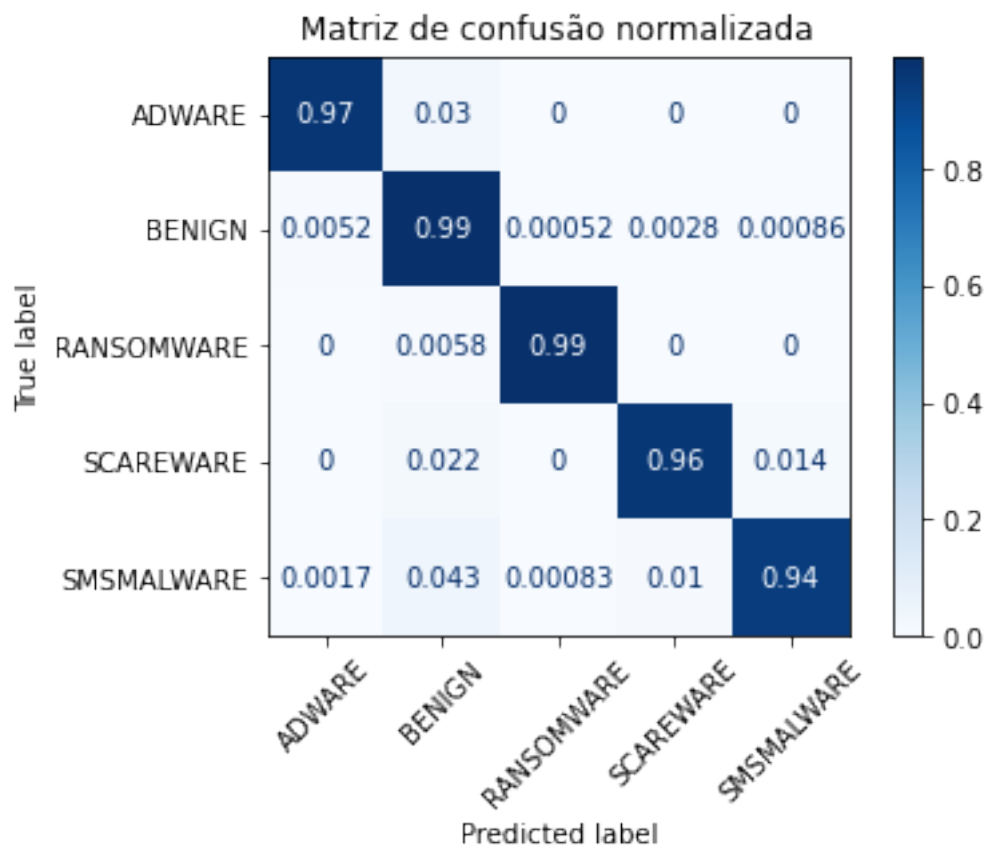




Acurácia: 0.979

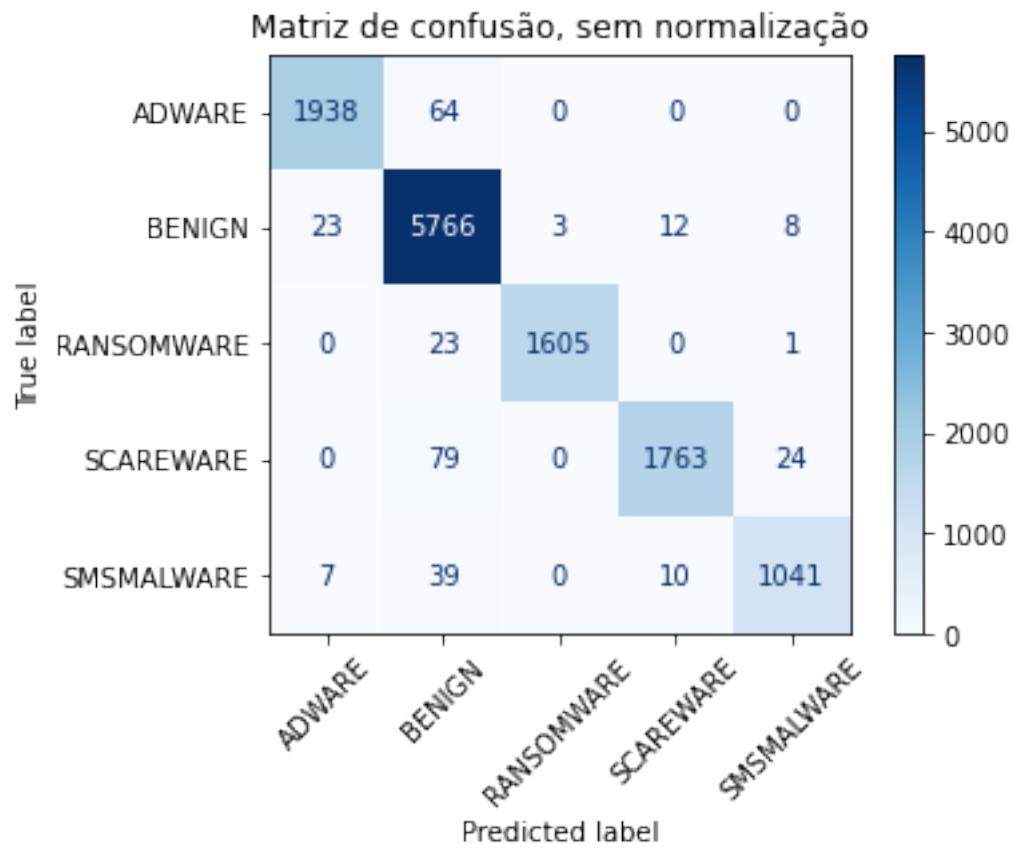
Erro: 0.021

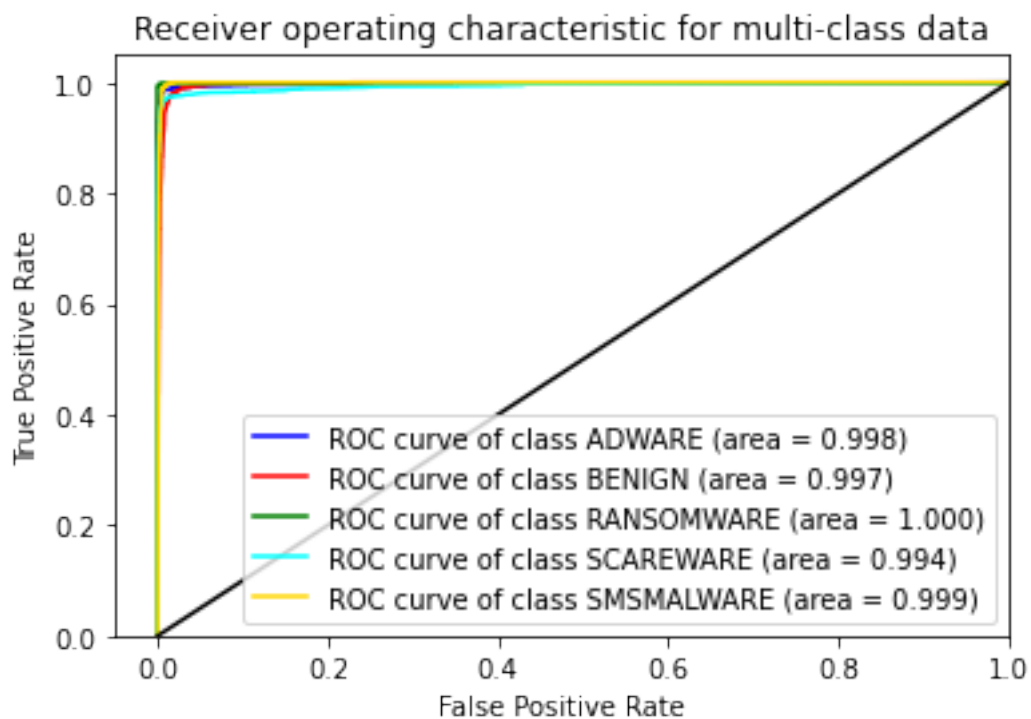
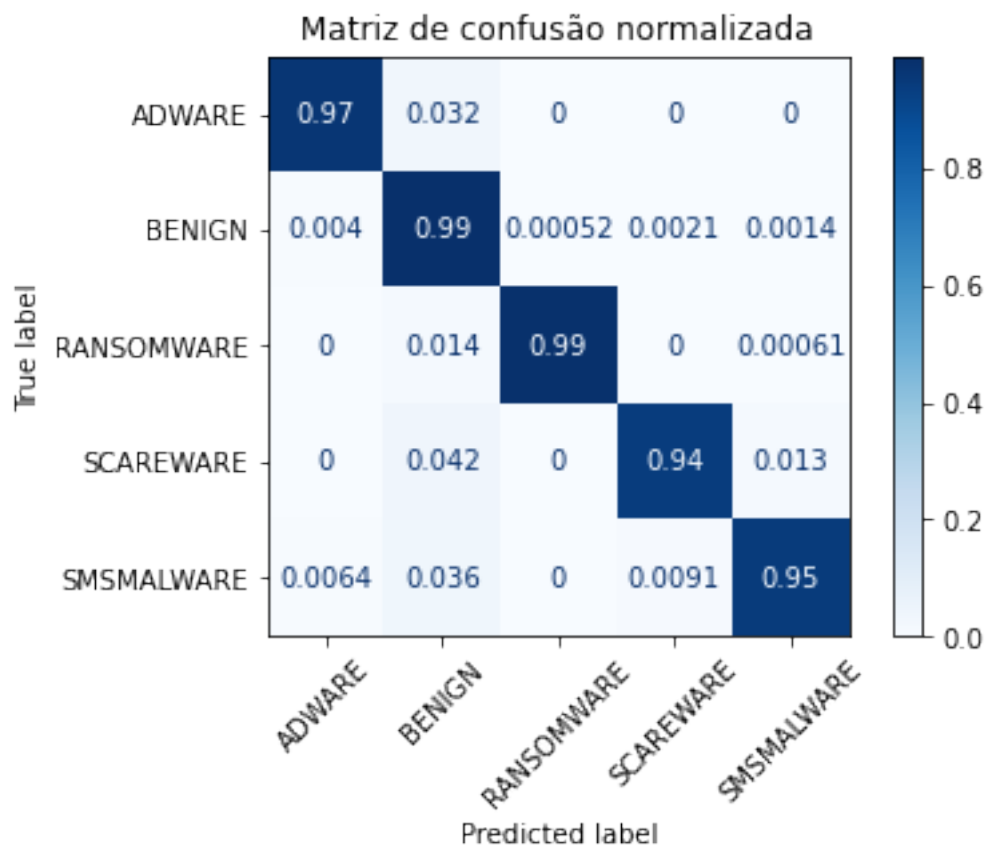




Acurácia: 0.976

Erro: 0.024





Média: 0.9783340963425958

```
[74]: media_acc = sum(scores)/len(scores)
media_erro = 1 - media_acc
print("Acurácia média:", media_acc)
print("Erro médio:", media_erro)
```

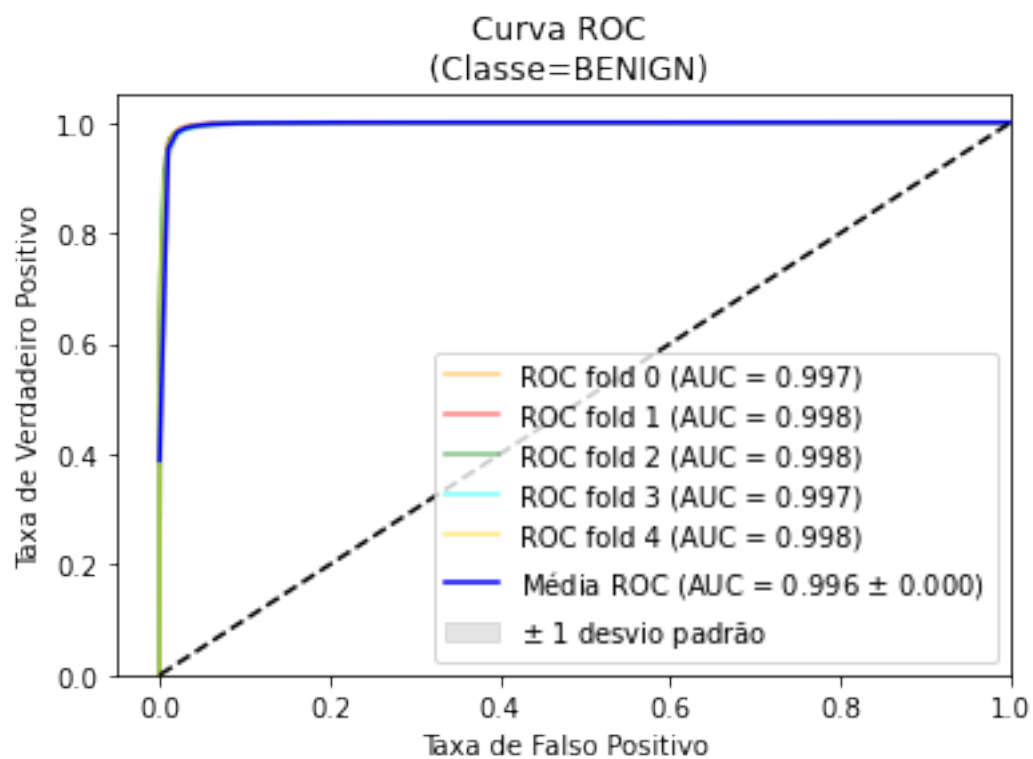
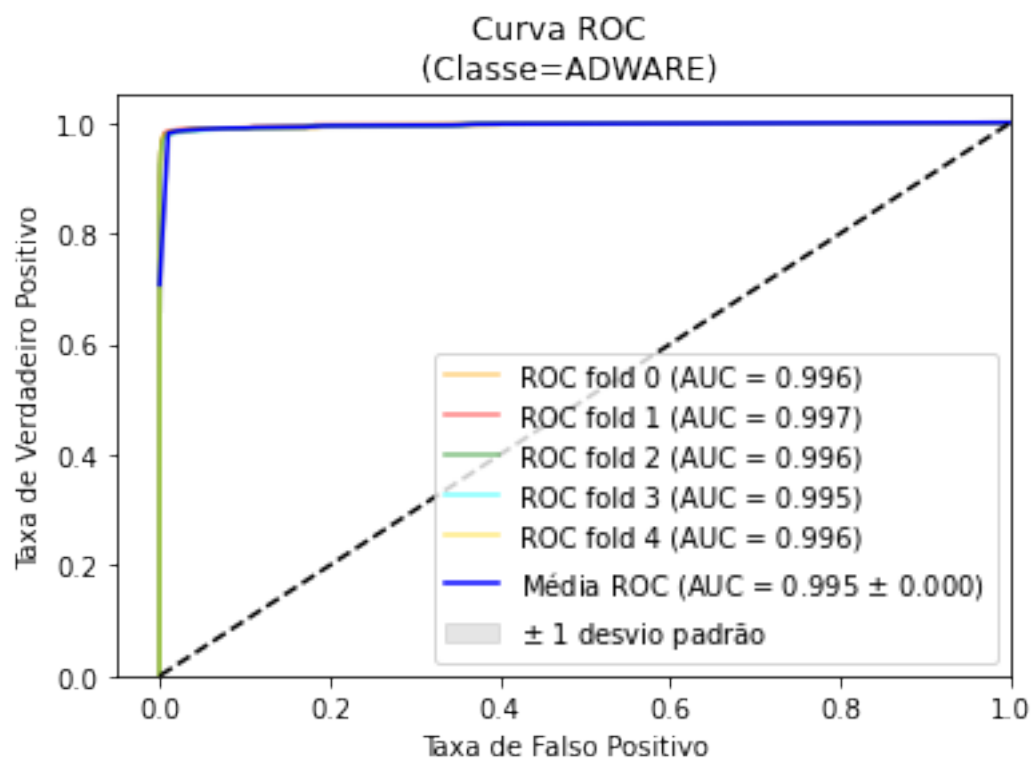
Acurácia média: 0.9783340963425958

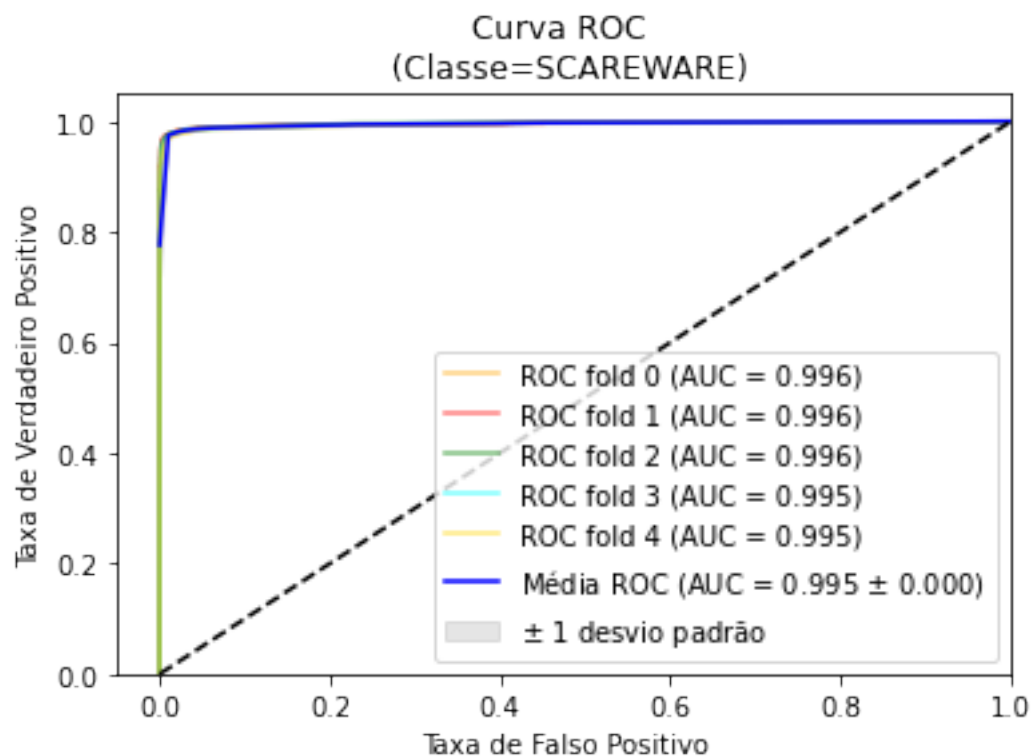
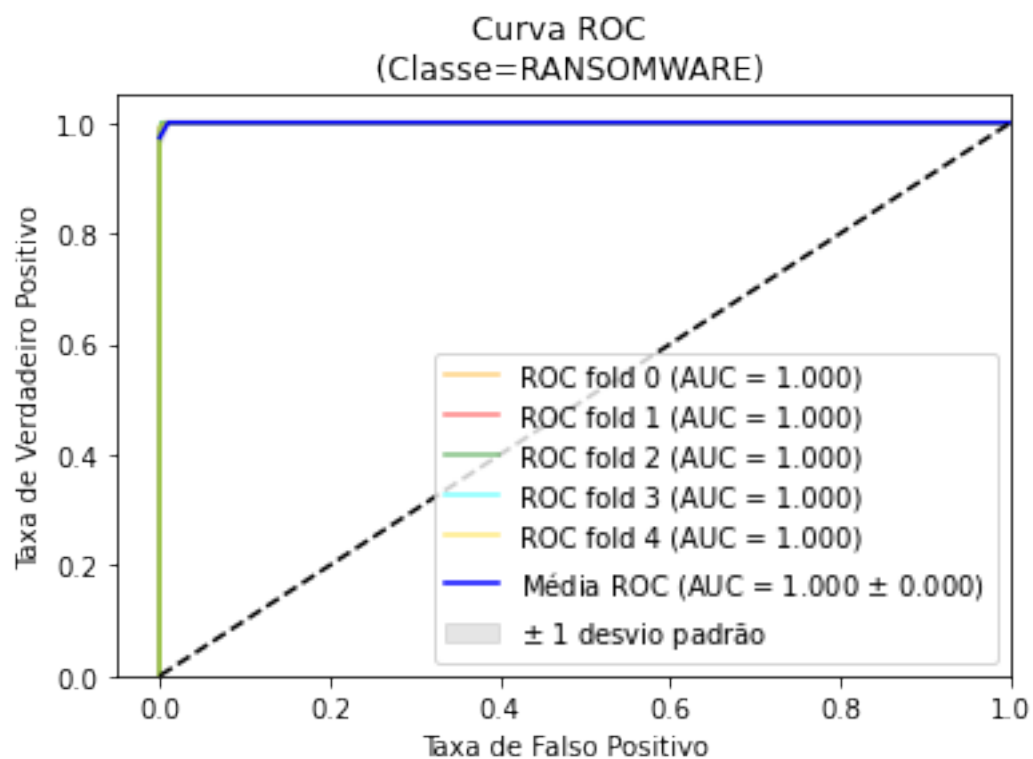
Erro médio: 0.021665903657404173

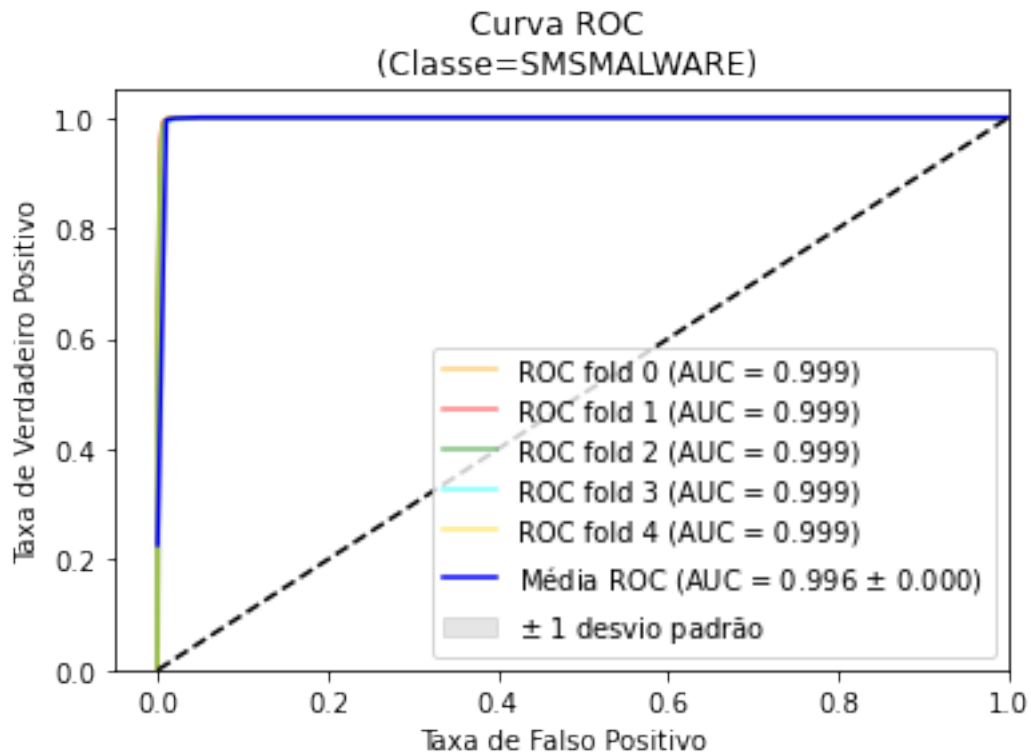
```
[75]: scores_array = np.array(scores)
models_array = np.array(models)
folds_array = np.array(folds)
index = np.where(scores_array == max(scores_array))
melhor_model_RF_k_fold = models_array[index]
RF_k_fold = folds_array[index[0]]
```

```
[76]: # Guarda o modelo gerado
dic_melhores_modelos['melhor_val_cruz_rf'].append(melhor_model_RF_k_fold)
dic_melhores_modelos['melhor_val_cruz_rf'].append(media_acc)
dic_melhores_modelos['melhor_val_cruz_rf'].append(RF_k_fold)
```

```
[36]: # Curva ROC por classe
funcoes_uteis.plot_roc_k_fold(clf_rf, X_treinamento_norm, y_treinamento_norm, 5)
```







3.4.4 MLP (*Multi-layer Perceptron*)

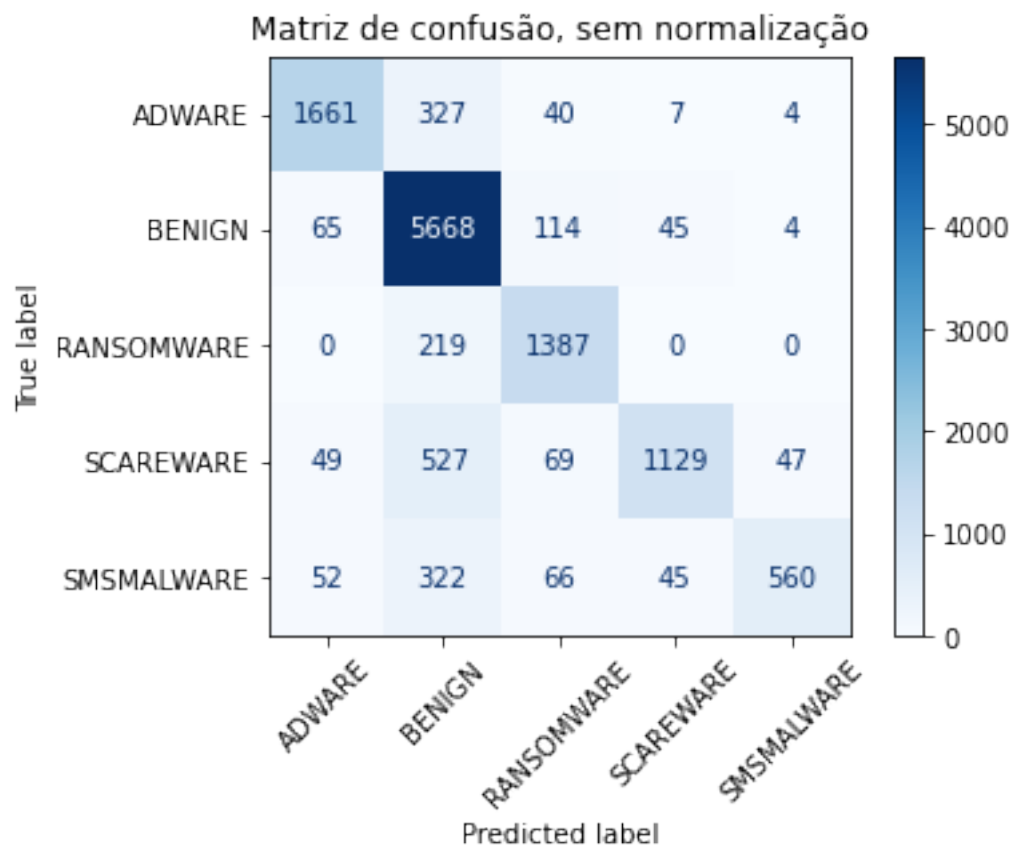
- Neste algoritmo foram utilizados os mesmos hiperparâmetros da etapa *percentage split*. Portanto o experimento para o MLP será executado com os seguintes hiperparâmetros e dados:
 - Dados: Normalizados
 - Configuração de camadas ocultas: (100,100)

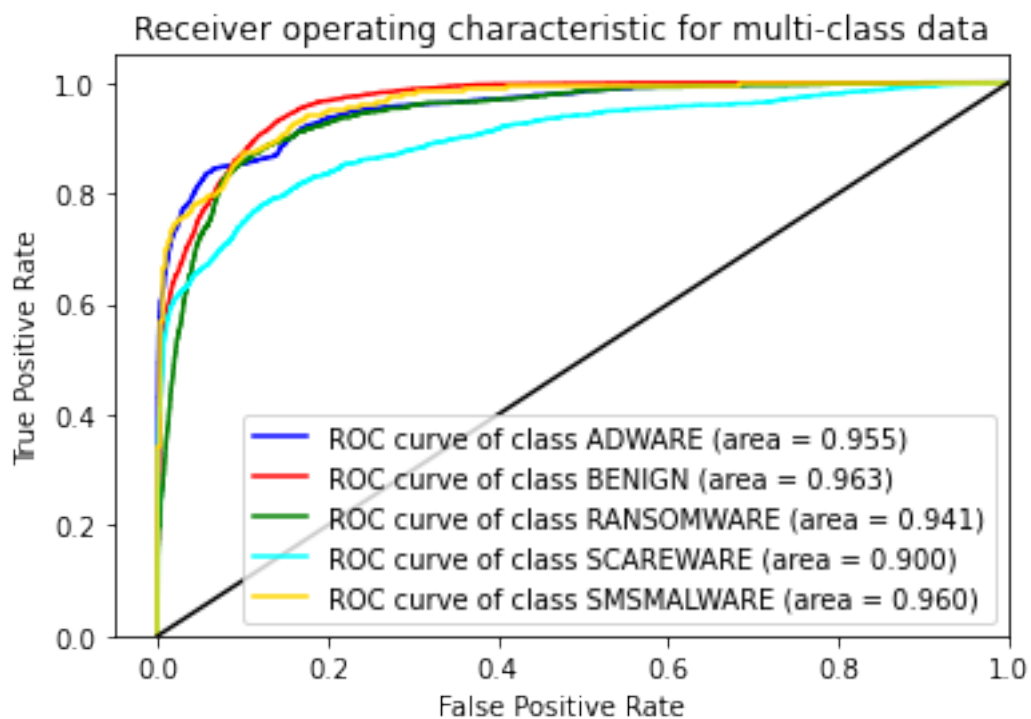
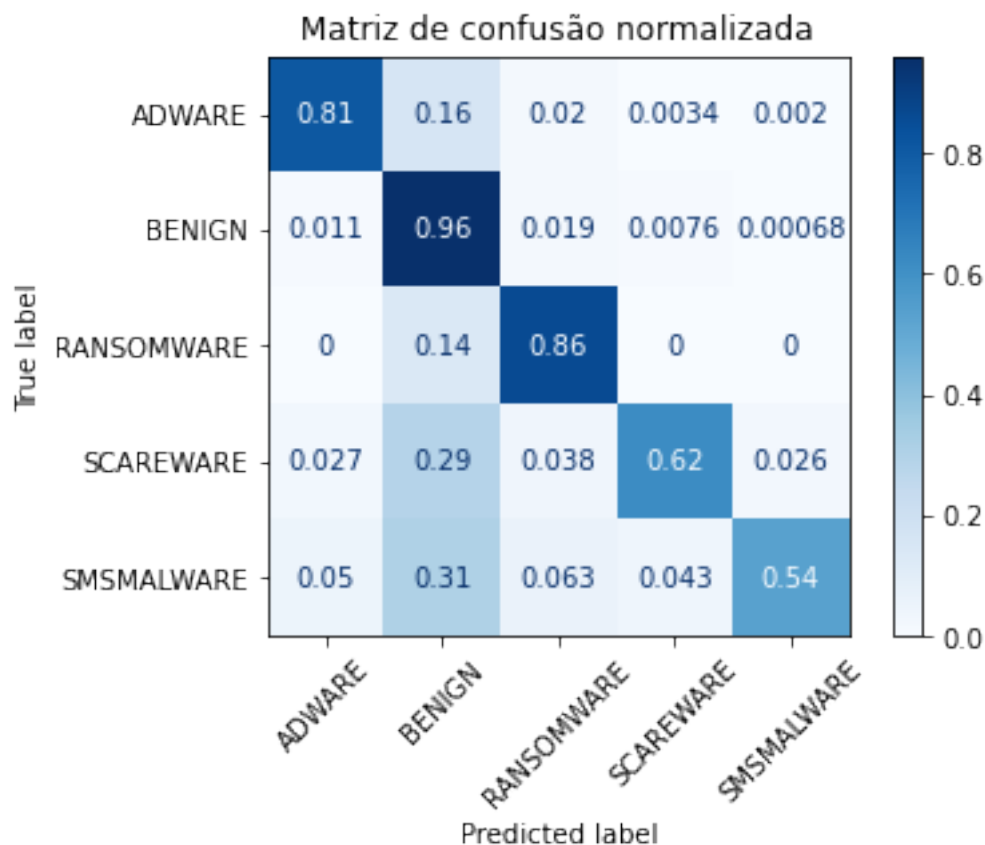
```
[60]: clf_mlp = MLPClassifier(hidden_layer_sizes=(100,100), random_state=1)

scores, models, folds = funcoes_uteis.k_fold_train(clf_mlp,
↪df_treinamento_norm, clf_name='MLP')
```

Acurácia: 0.839

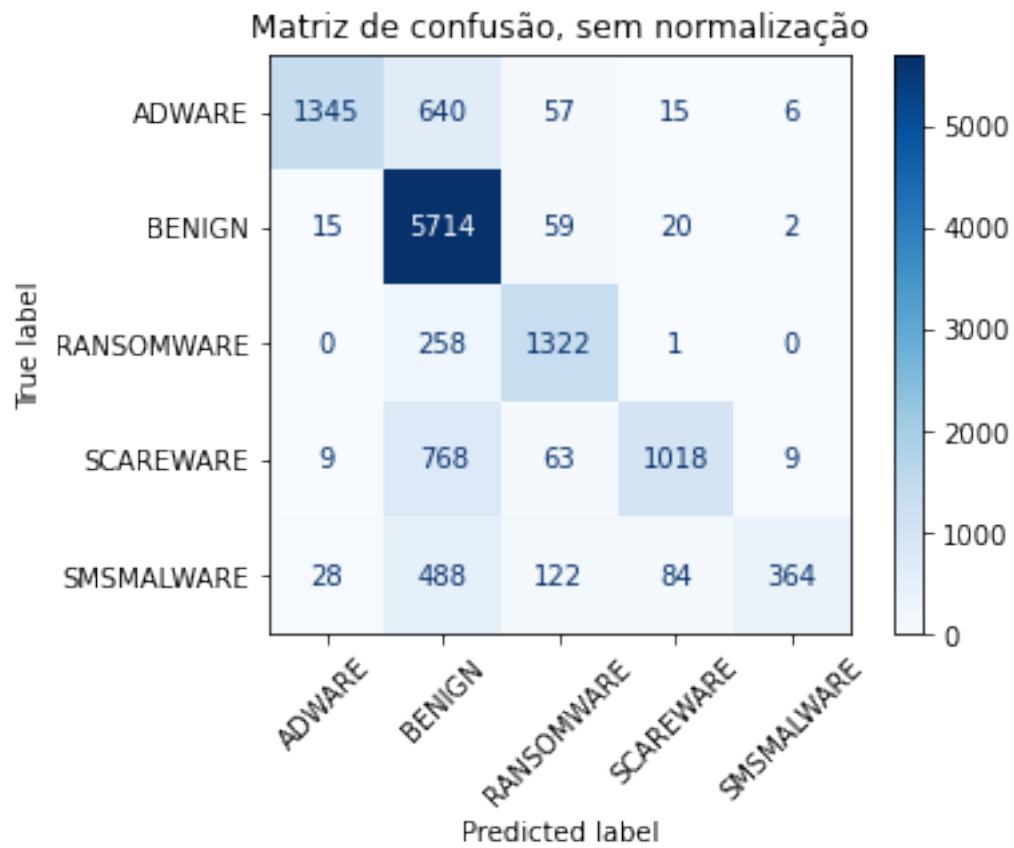
Erro: 0.161

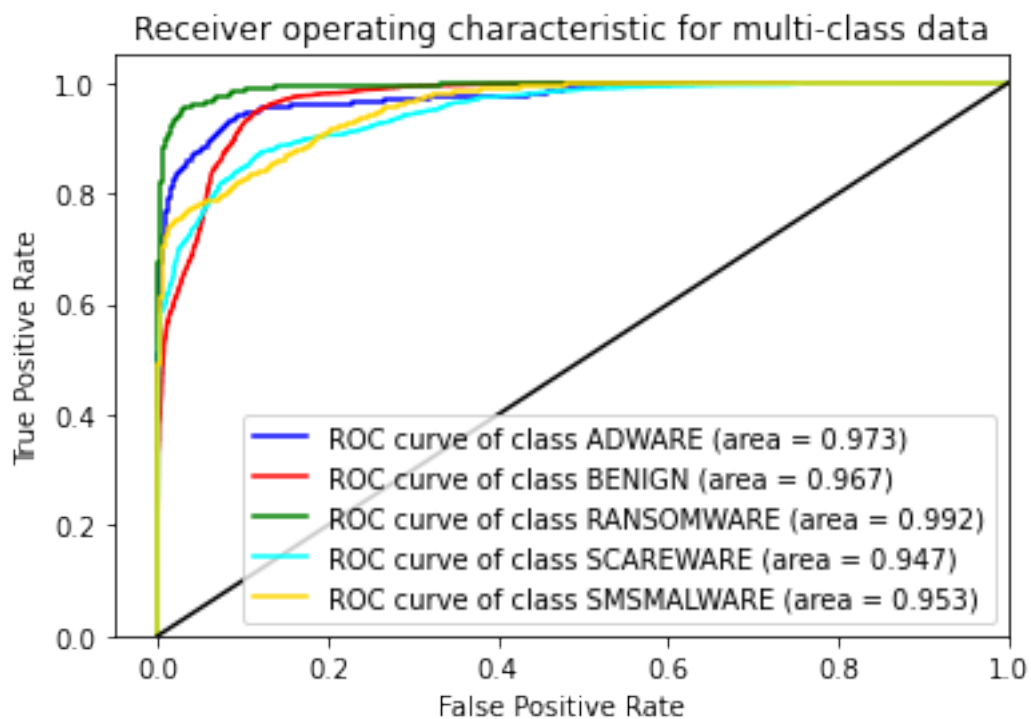
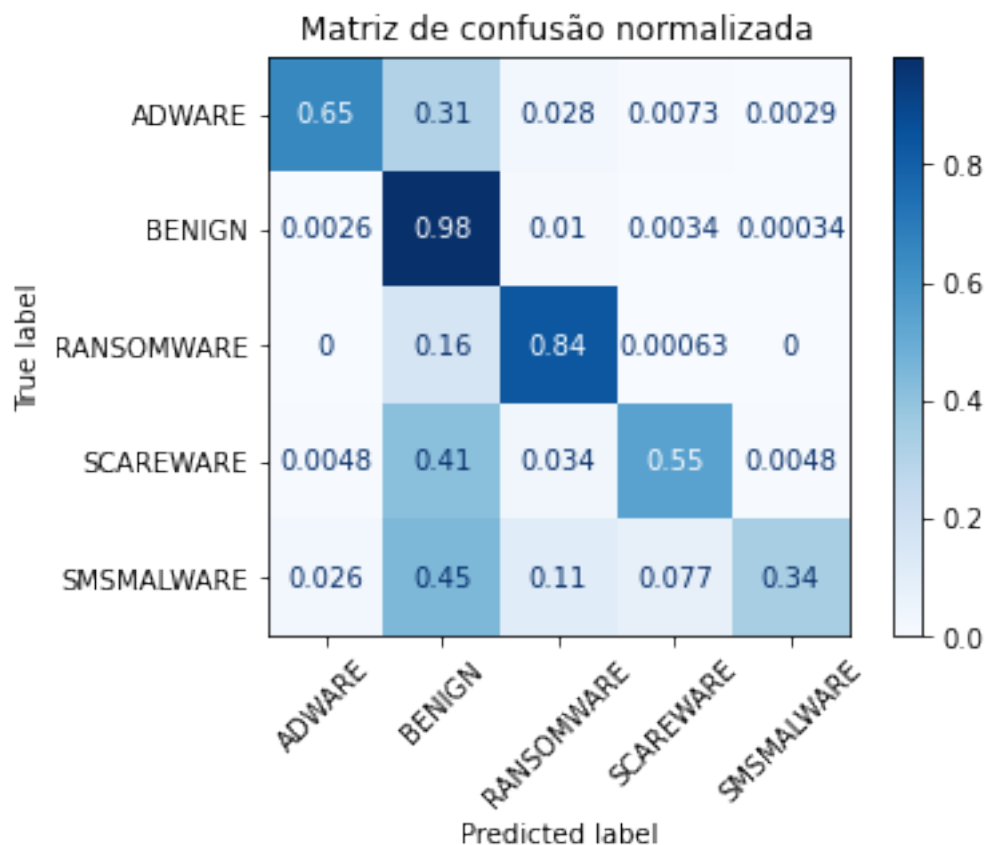




Acurácia: 0.787

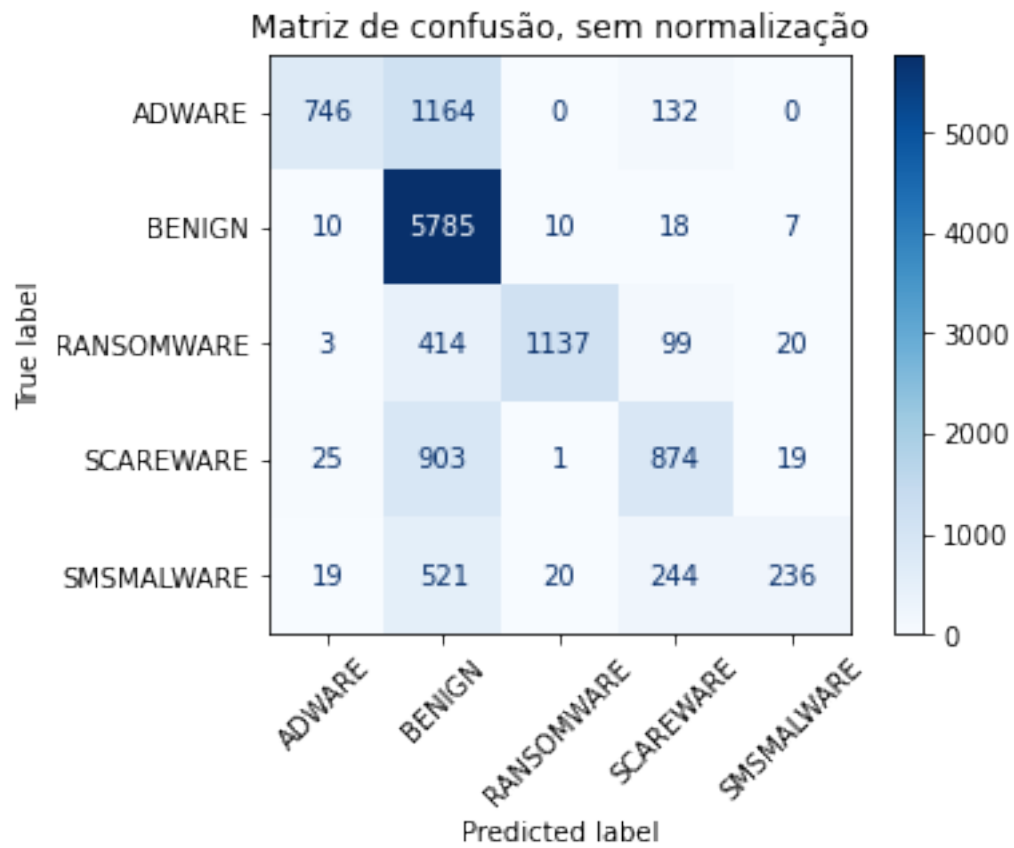
Erro: 0.213

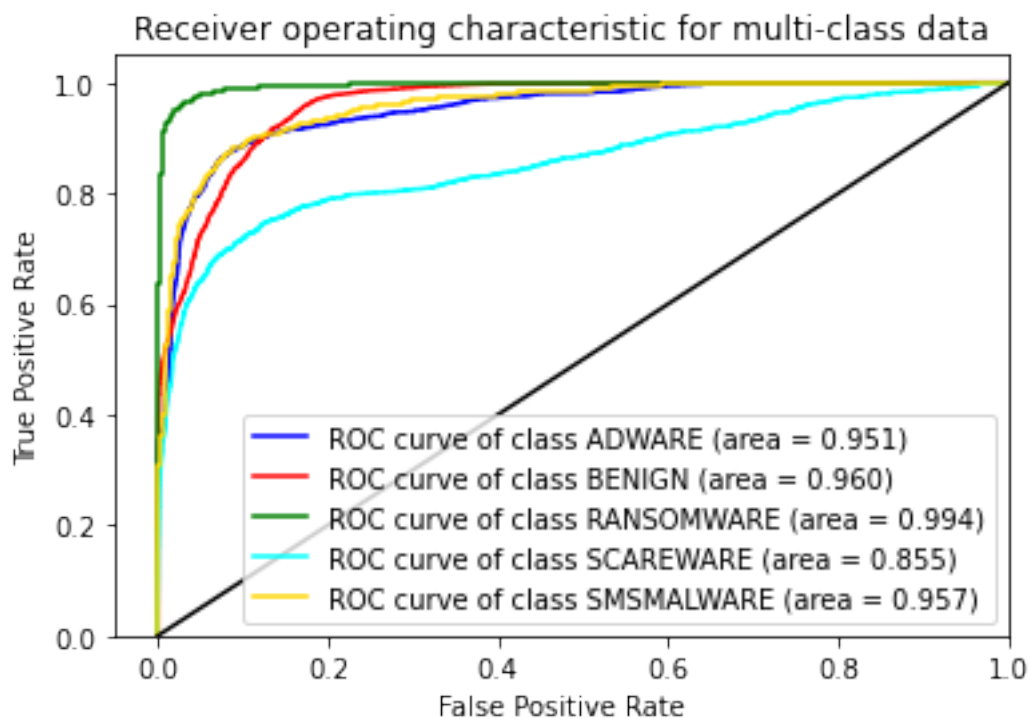
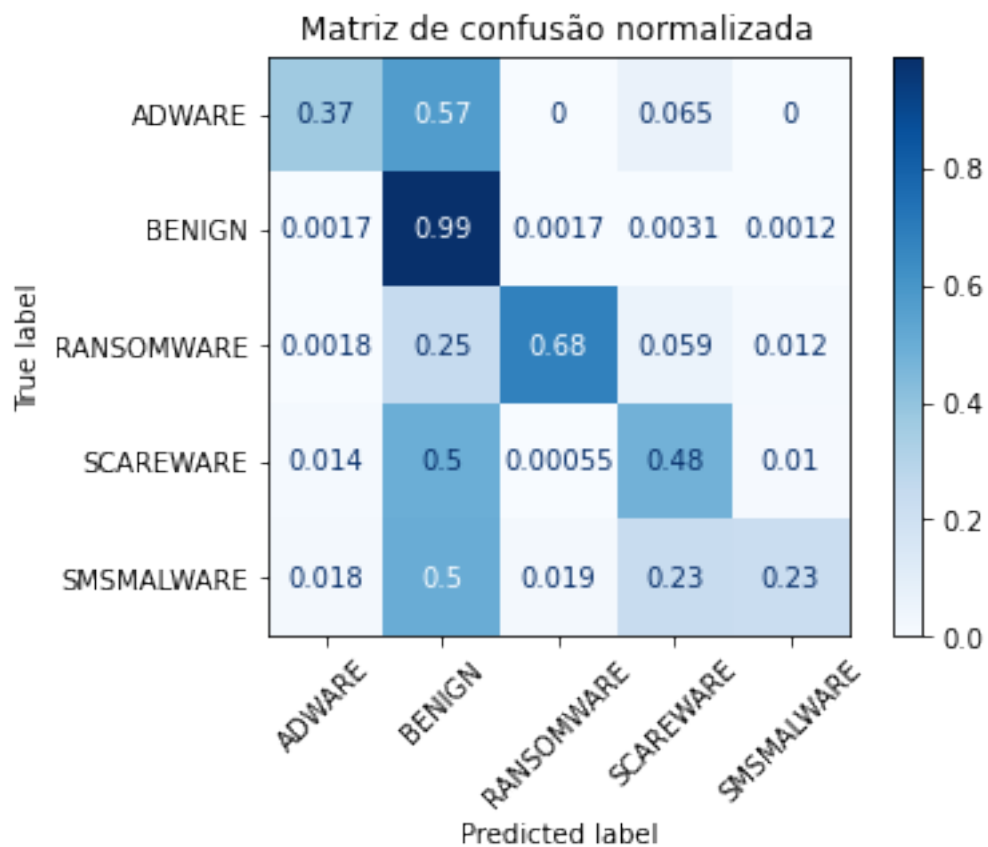




Acurácia: 0.708

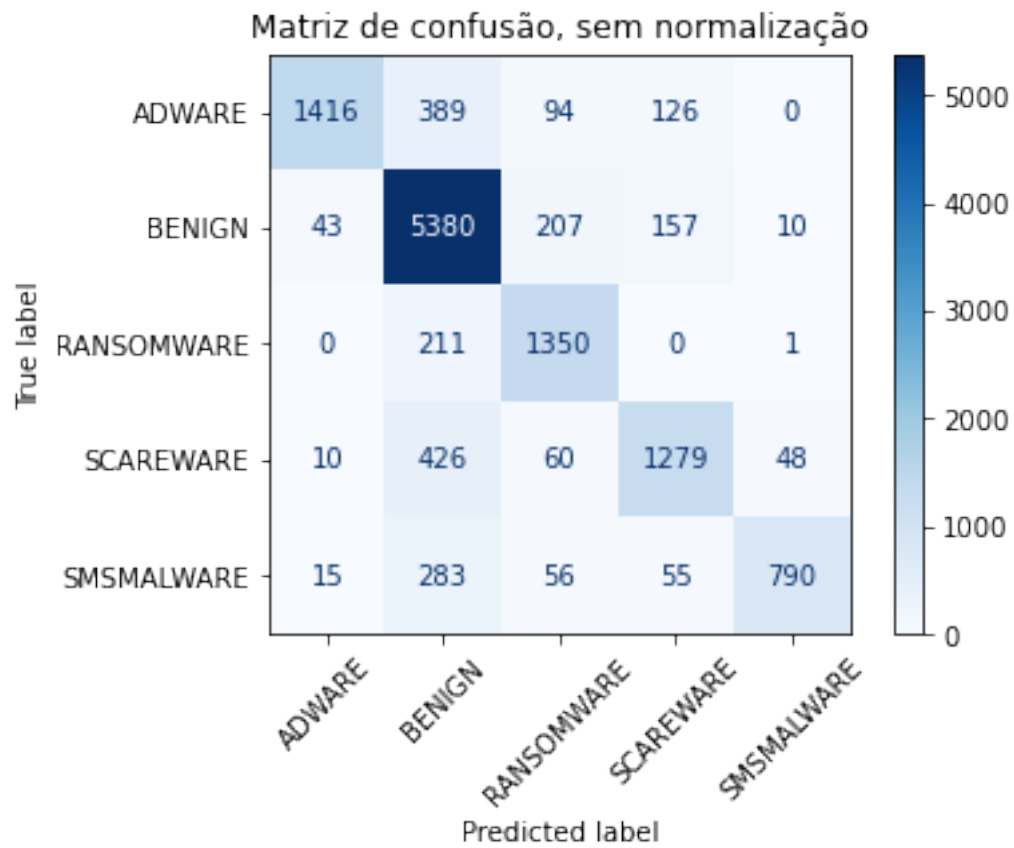
Erro: 0.292

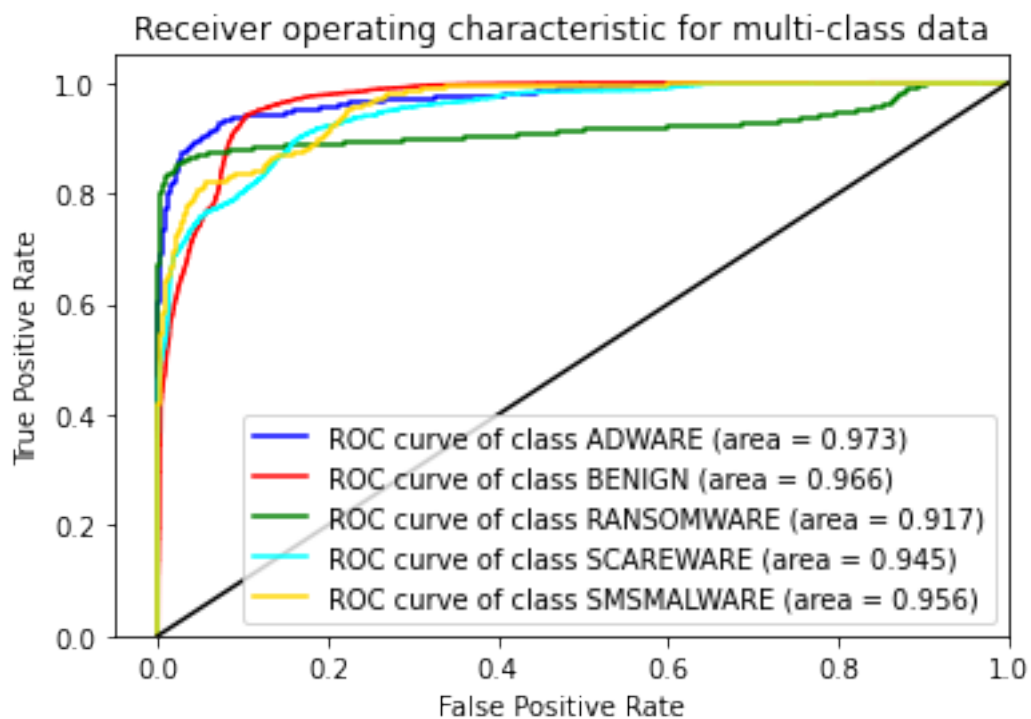
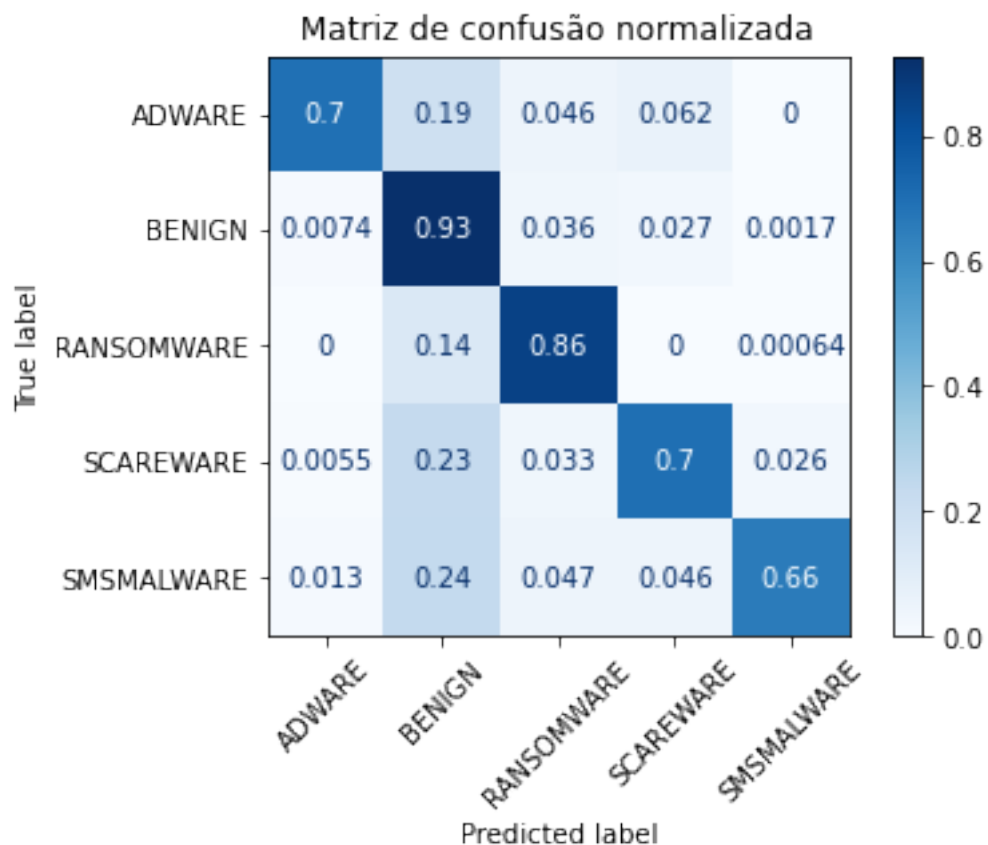




Acurácia: 0.823

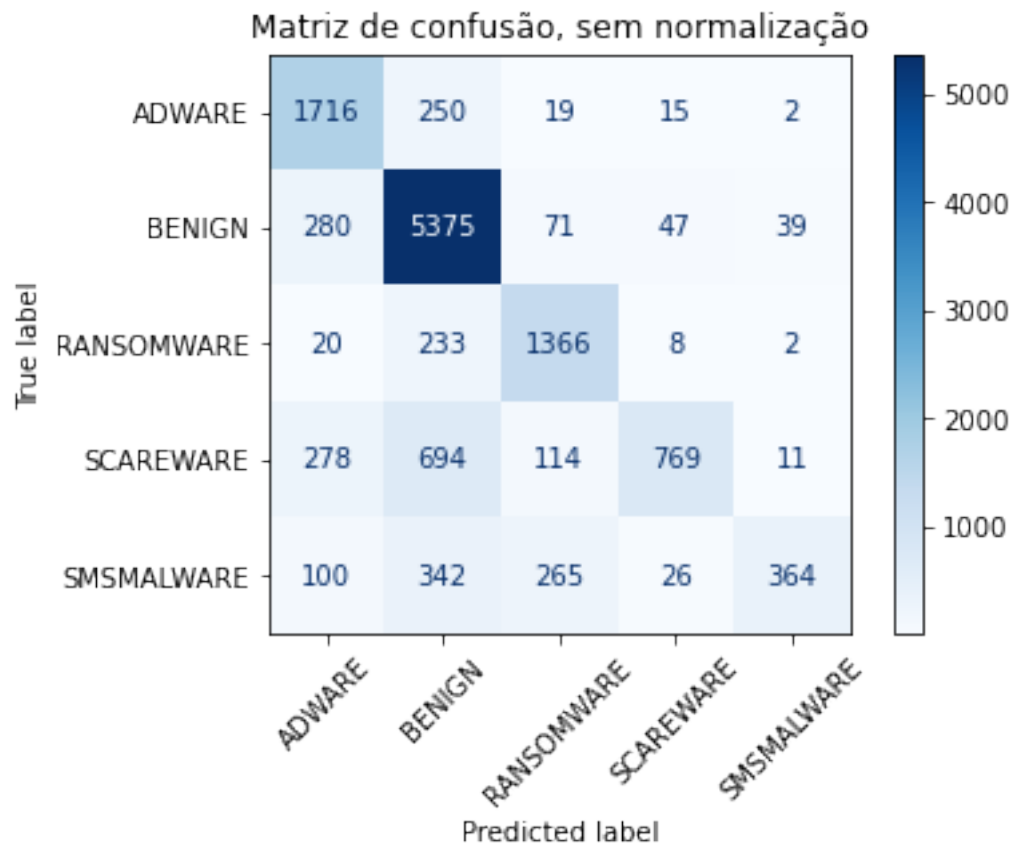
Erro: 0.177

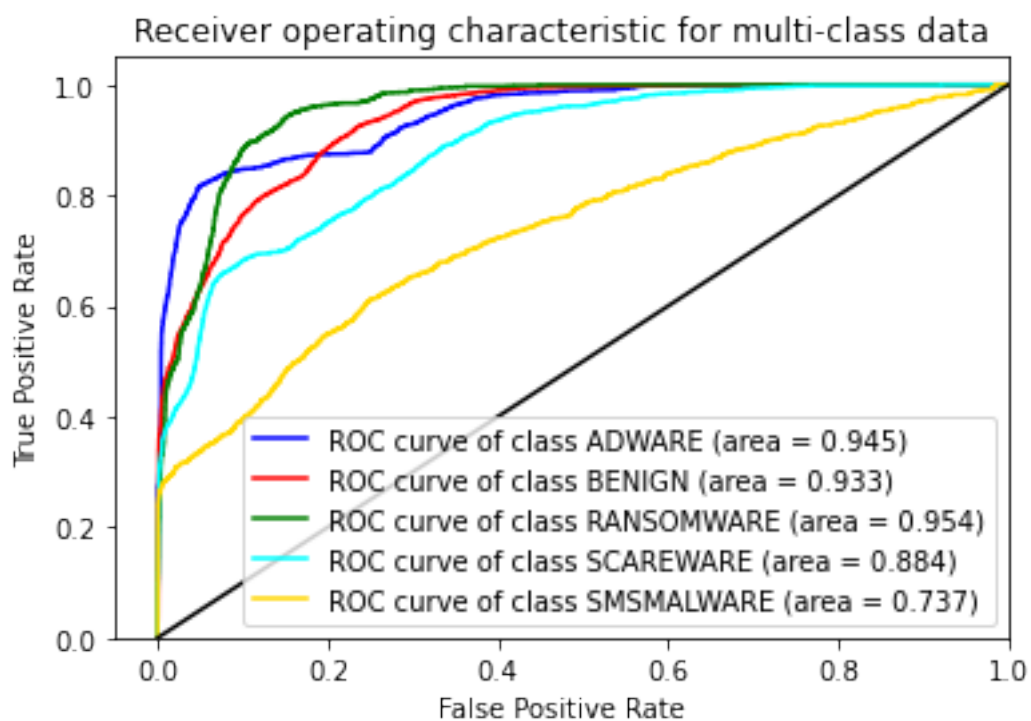
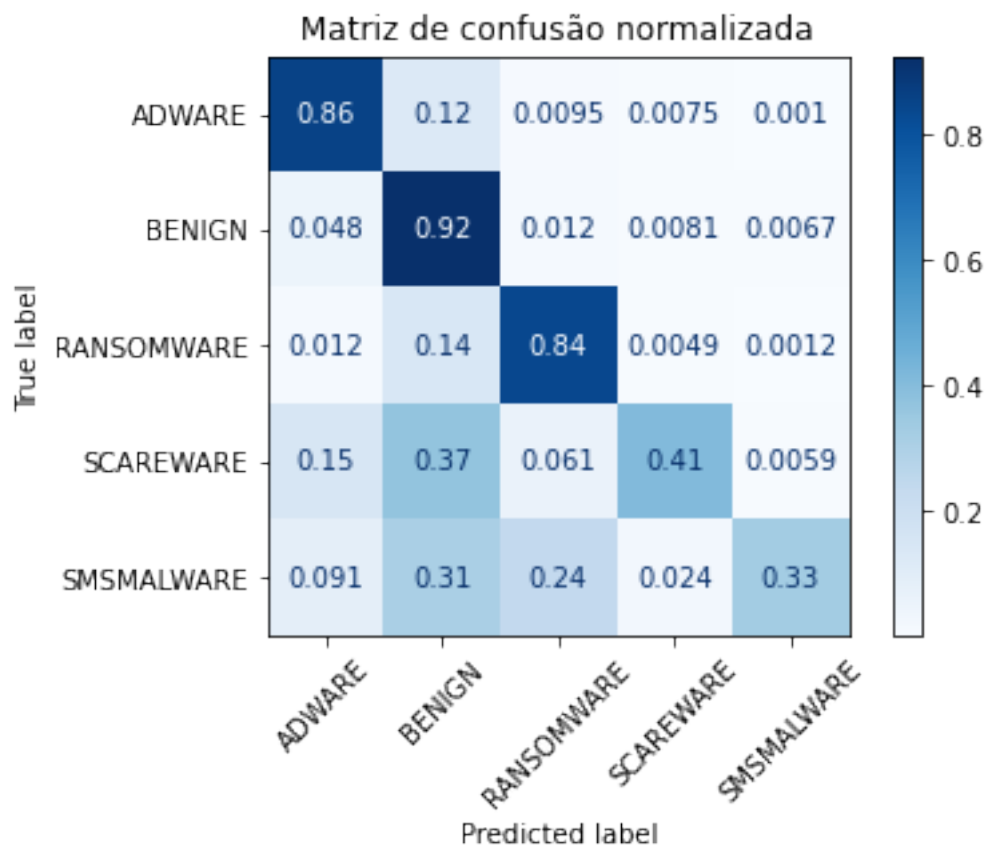




Acurácia: 0.773

Erro: 0.227





Média: 0.7858885533161174

```
[61]: media_acc = sum(scores)/len(scores)
      media_erro = 1 - media_acc
      print("Acurácia média:", media_acc)
      print("Erro médio:", media_erro)
```

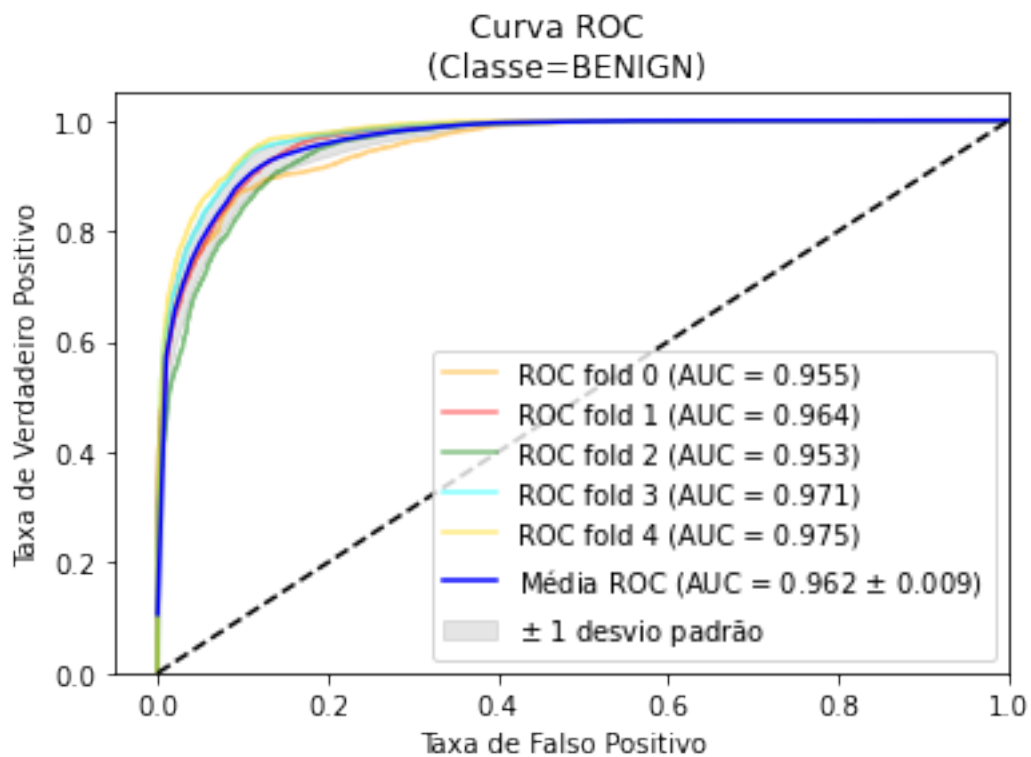
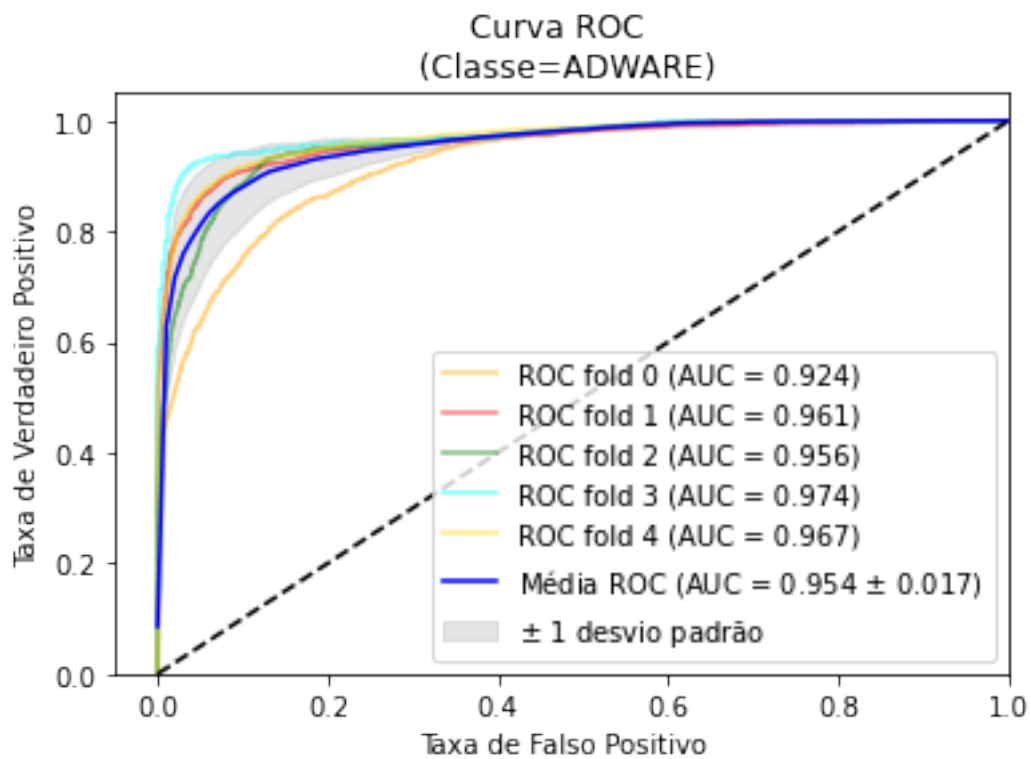
Acurácia média: 0.7858885533161174

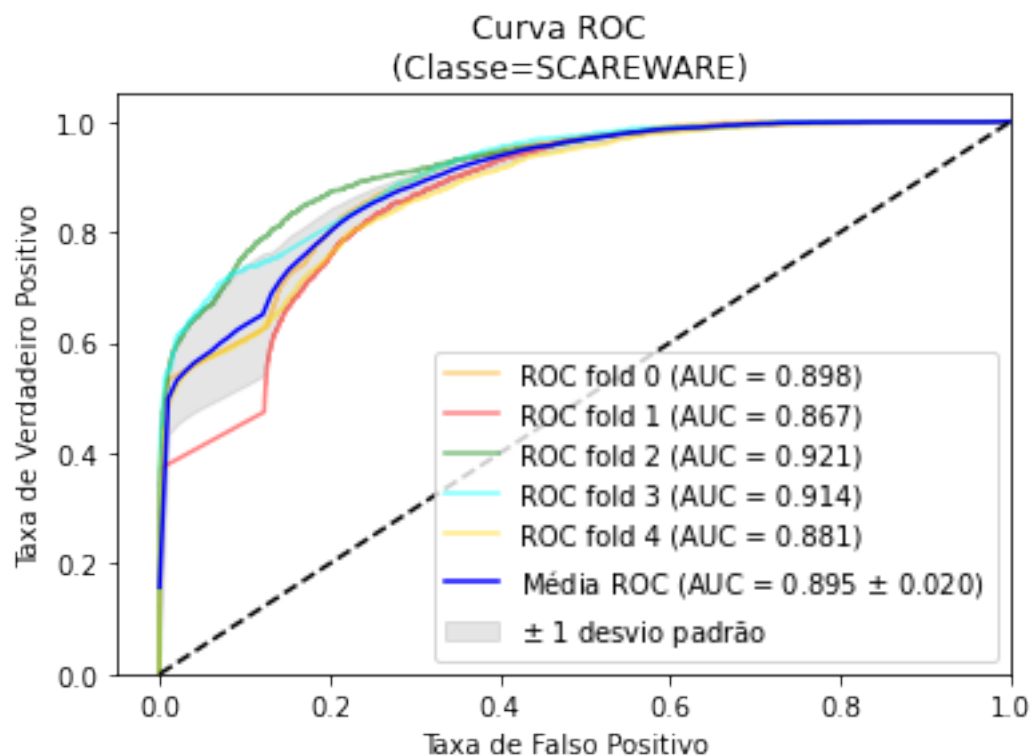
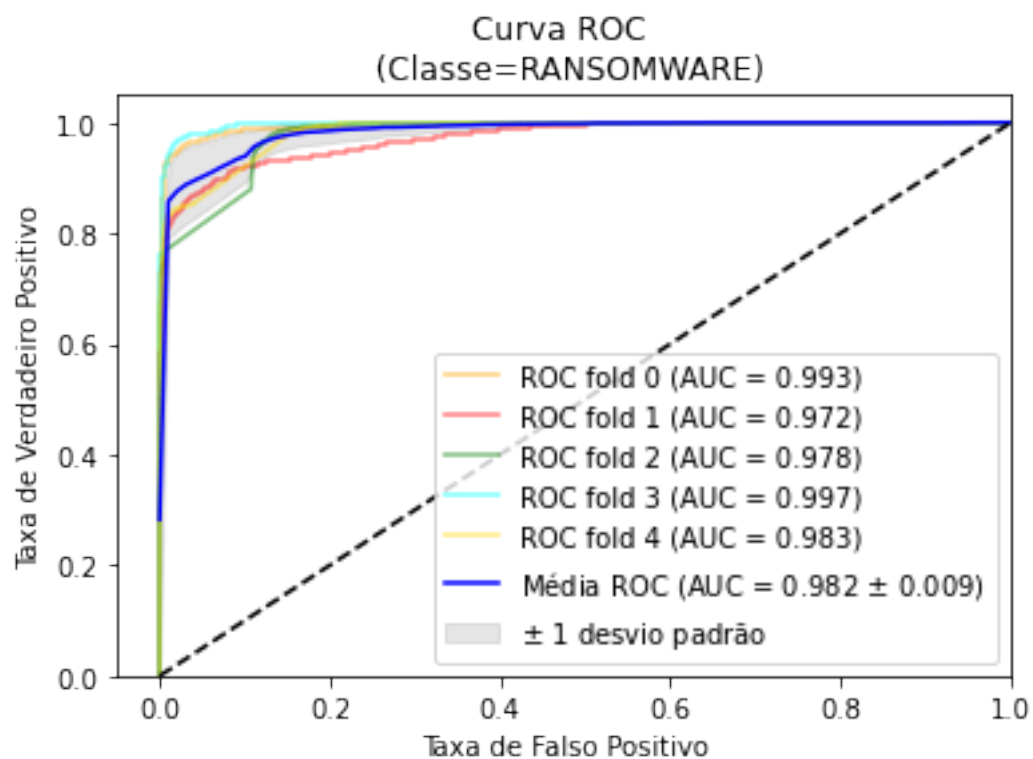
Erro médio: 0.2141114466838826

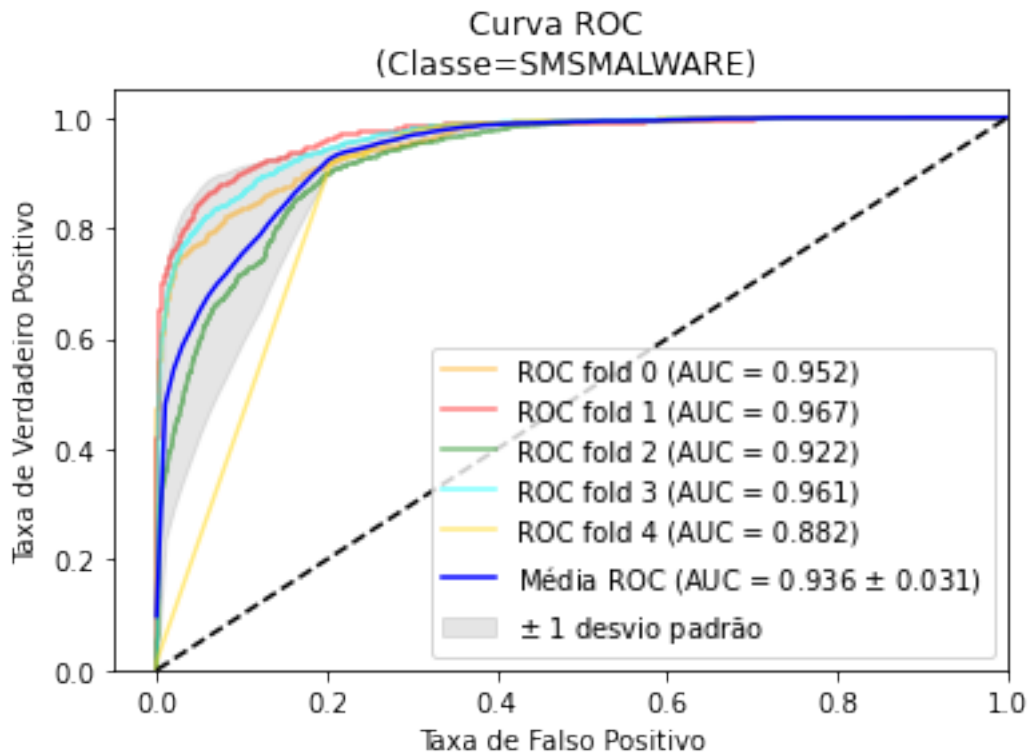
```
[76]: scores_array = np.array(scores)
      models_array = np.array(models)
      folds_array = np.array(folds)
      index = np.where(scores_array == max(scores_array))
      melhor_model_mlp_k_fold = models_array[index]
      MLP_k_fold = folds_array[index[0]]
```

```
[63]: # Guarda o modelo gerado
      dic_melhores_modelos['melhor_val_cruz_mlp'].append(melhor_model_mlp_k_fold)
      dic_melhores_modelos['melhor_val_cruz_mlp'].append(media_acc)
      dic_melhores_modelos['melhor_val_cruz_mlp'].append(MLP_k_fold)
```

```
[43]: # Curva ROC por classe
      funcoes_uteis.plot_roc_k_fold(clf_mlp, X_treinamento_norm, y_treinamento_norm,
      ↪5)
```





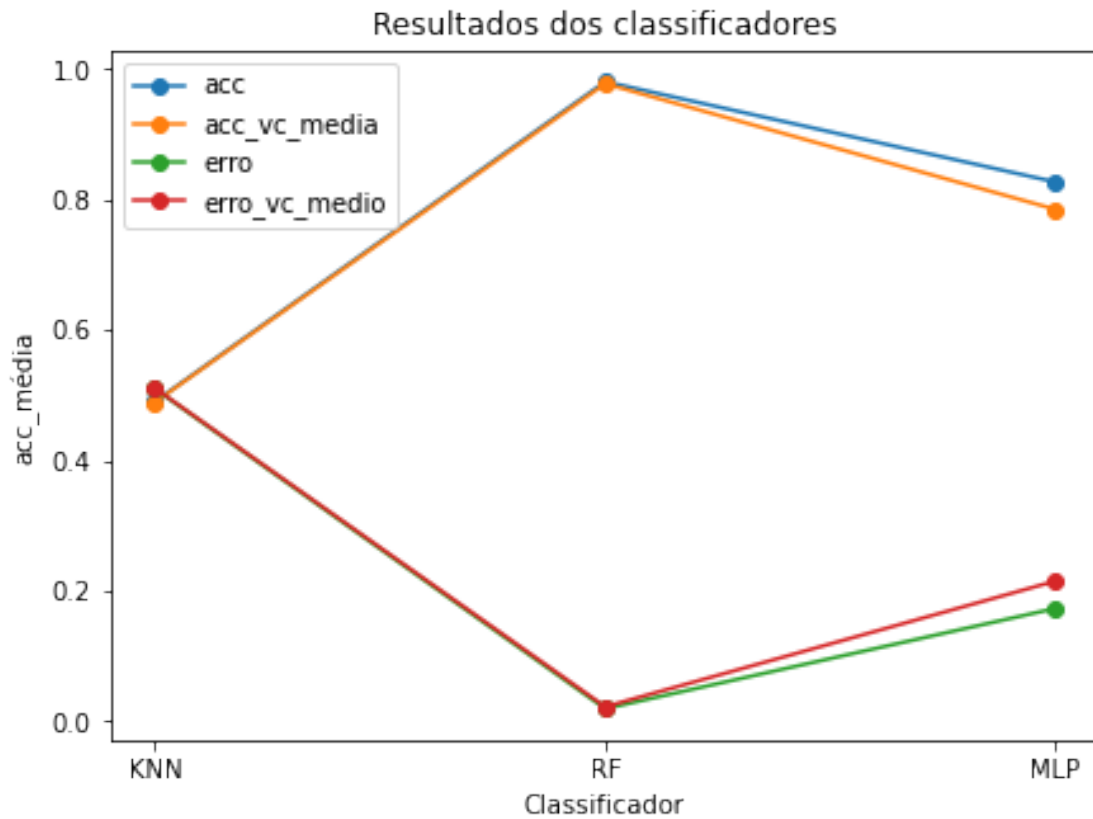


3.4.5 Comparação de Resultados e discussão

- É possível observar pelos experimentos com validação cruzada que os resultados são similares aos com *percentage split*, o gráfico abaixo demonstra claramente a similaridade. Esta similaridade permite-se dizer que os modelos gerados apenas com *percentage split* tendem a ser confiáveis. Novamente o MLP apresentou dificuldades em prever SMSMALWARE e SCAREWARE. Esta dificuldade fica fácil de se observar nestes experimentos, pois foi possível traçar a curva ROC por classe para cada *fold* e para estas duas classes a curva é mais próxima ao centro do gráfico.
- Entre todos os modelos gerados, o *Random Forest* conseguiu atingir os melhores resultados. A dificuldade que o MLP apresenta sobre as classes SMSMALWARE e SCAREWARE é superada pelo *Random Forest*, atingindo acertos de mais de 94/%, conforme pode ser visto pela matriz de confusão.

```
[164]: classificador_vc = classificador_split_test + ['KNN', 'RF', 'MLP', 'KNN', 'RF', 'MLP',]
acuracias_medias_vc = acuracias_split_test + [0.489, 0.978, 0.785, 1-0.489, 1-0.978, 1-0.785]
divisao_vc = divisao_split_test + ['acc_vc_media', 'acc_vc_media', 'acc_vc_media', 'erro_vc_medio', 'erro_vc_medio', 'erro_vc_medio']
```

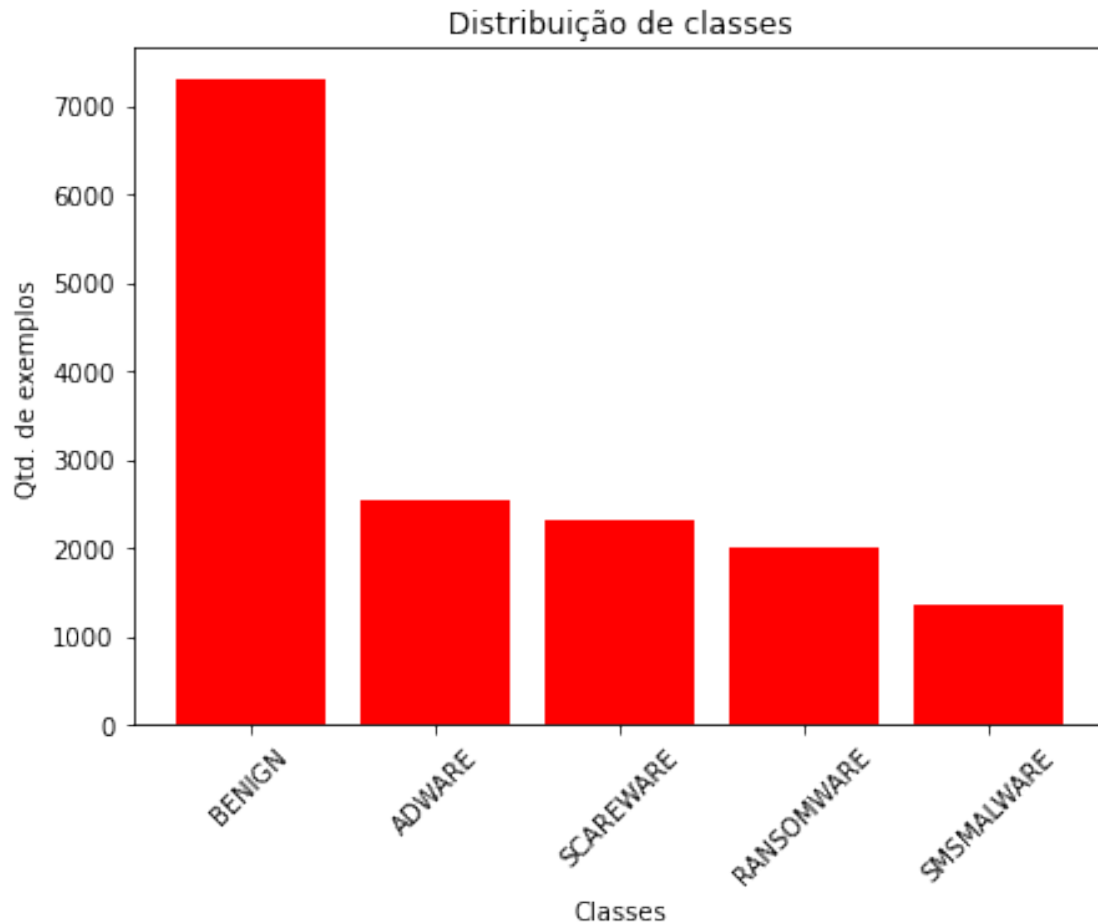
```
# Exibe os resultados
funcoes_uteis.plot_resultados(classificador_vc, 'Classificador',
                              acuracias_medias_vc, 'acc_média', divisao_vc,
                              'Resultados dos classificadores', figsize=(7,5))
```



4 Validação dos modelos

- Nesta etapa foram calculadas as taxas de acerto, erro e exibidos a matriz de confusão e curva ROC dos melhores modelos gerados pelas etapas anteriores, utilizando-se para isso os dados de validação.
- Nas etapas anteriores os dados normalizados foram escolhidos por alcançarem os melhores resultados, portanto, nesta etapa, também serão utilizados os dados normalizados.

```
[55]: # Distribuição das classes
funcoes_uteis.plot_distribuicao_classes(df_validacao_norm, 'Label')
```

4.1 Preparação dos dados para validação, normalizados e não normalizados

```
[156]: # Remove a coluna NOME_APP
df_validacao_norm = df_validacao_norm.drop('NOME_APP', axis=1)
df_validacao = df_validacao.drop('NOME_APP', axis=1)
```

```
[157]: # Dados de validação
X_validacao_norm = df_validacao_norm.iloc[:,0:-1].values
y_validacao_norm = df_validacao_norm.iloc[:,-1].values

X_validacao = df_validacao.iloc[:,0:-1].values
y_validacao = df_validacao.iloc[:,-1].values
```

4.2 Melhores modelos gerados com *Percentage split*

4.2.1 KNN (*K-nearest neighbors algorithm*)

```
[112]: # Obtém o modelo gerado
modelo_knn_ps = dic_melhores_modelos['split_perc_knn'][0]
y_pred = modelo_knn_ps.predict(X_validacao_norm)
print(classification_report(y_pred, y_validacao_norm, zero_division=True))
```

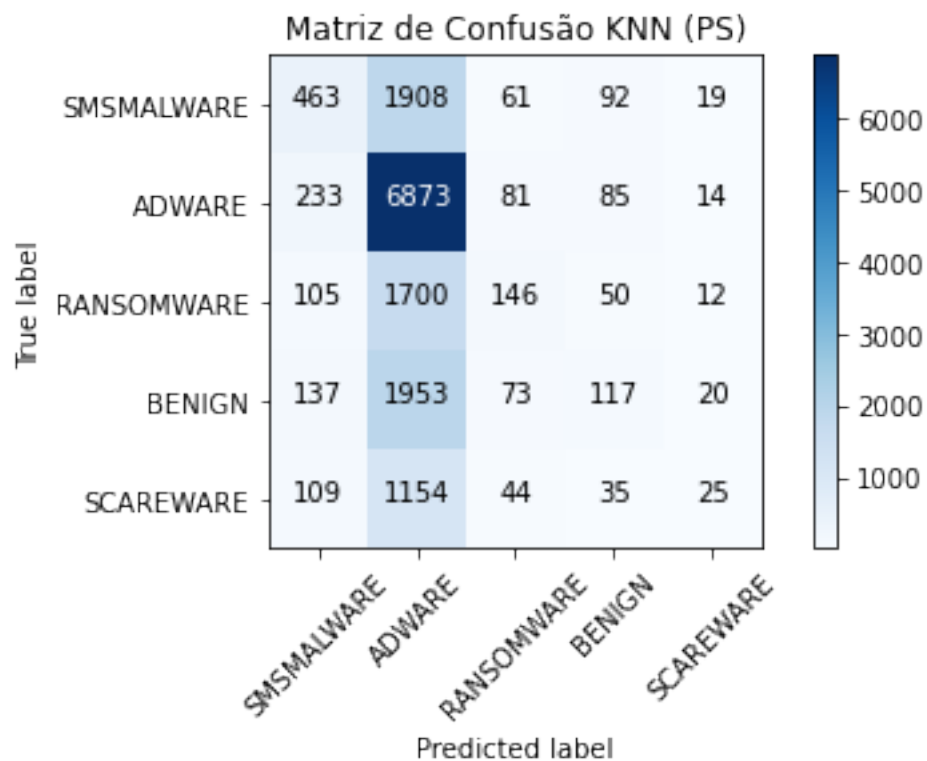
	precision	recall	f1-score	support
ADWARE	0.18	0.44	0.26	1047
BENIGN	0.94	0.51	0.66	13588
RANSOMWARE	0.07	0.36	0.12	405
SCAREWARE	0.05	0.31	0.09	379
SMSMALWARE	0.02	0.28	0.03	90
accuracy			0.49	15509
macro avg	0.25	0.38	0.23	15509
weighted avg	0.84	0.49	0.60	15509

```
[113]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_validacao_norm, y_pred)
erro = 1 - accuracy
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

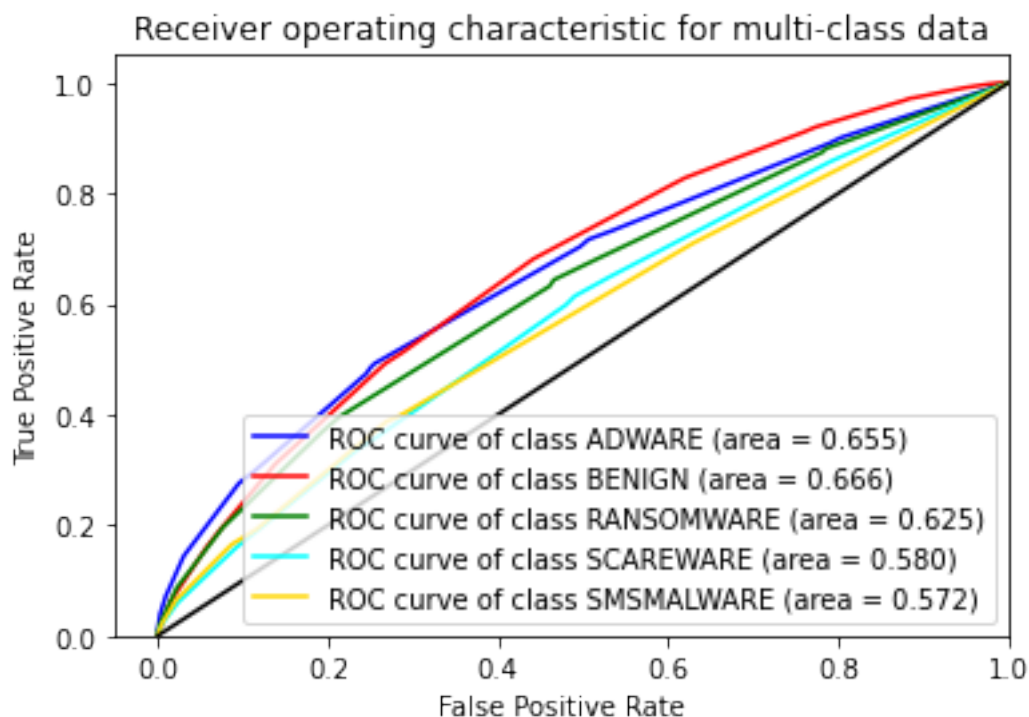
Acurácia: 0.492

Erro: 0.508

```
[114]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_validacao_norm, y_pred,
                                     set(y_validacao_norm),
                                     'Matriz de Confusão KNN (PS)')
```



```
[115]: # Exibe curva ROC
funcoes_uteis.plot_roc_curve(modelo_knn_ps, X_train_norm, y_train_norm,
                             X_validacao_norm, y_validacao_norm)
```



4.2.2 Random Forest

```
[116]: # Obtém o modelo gerado
modelo_rf_ps = dic_melhores_modelos['split_perc_rf'][0]
y_pred = modelo_rf_ps.predict(X_validacao_norm)
print(classification_report(y_pred, y_validacao_norm, zero_division=True))
```

	precision	recall	f1-score	support
ADWARE	0.97	0.99	0.98	2510
BENIGN	0.99	0.97	0.98	7481
RANSOMWARE	0.99	1.00	0.99	1986
SCAREWARE	0.95	0.99	0.97	2199
SMSMALWARE	0.94	0.96	0.95	1333
accuracy			0.98	15509
macro avg	0.97	0.98	0.98	15509
weighted avg	0.98	0.98	0.98	15509

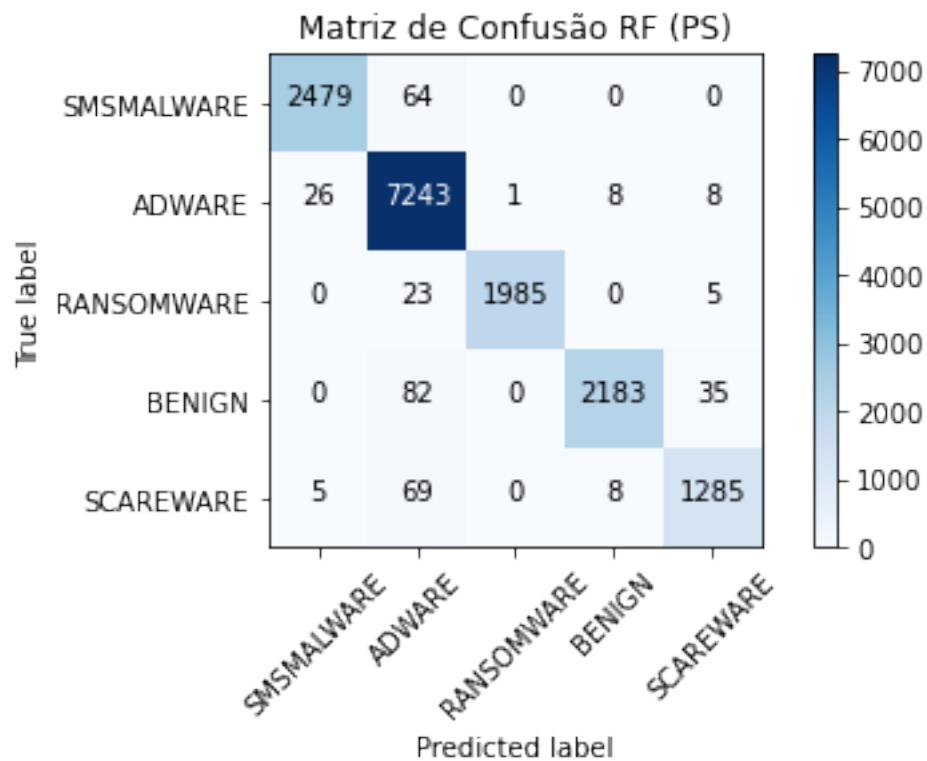
```
[117]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_validacao_norm, y_pred)
erro = 1 - accuracy
```

```
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

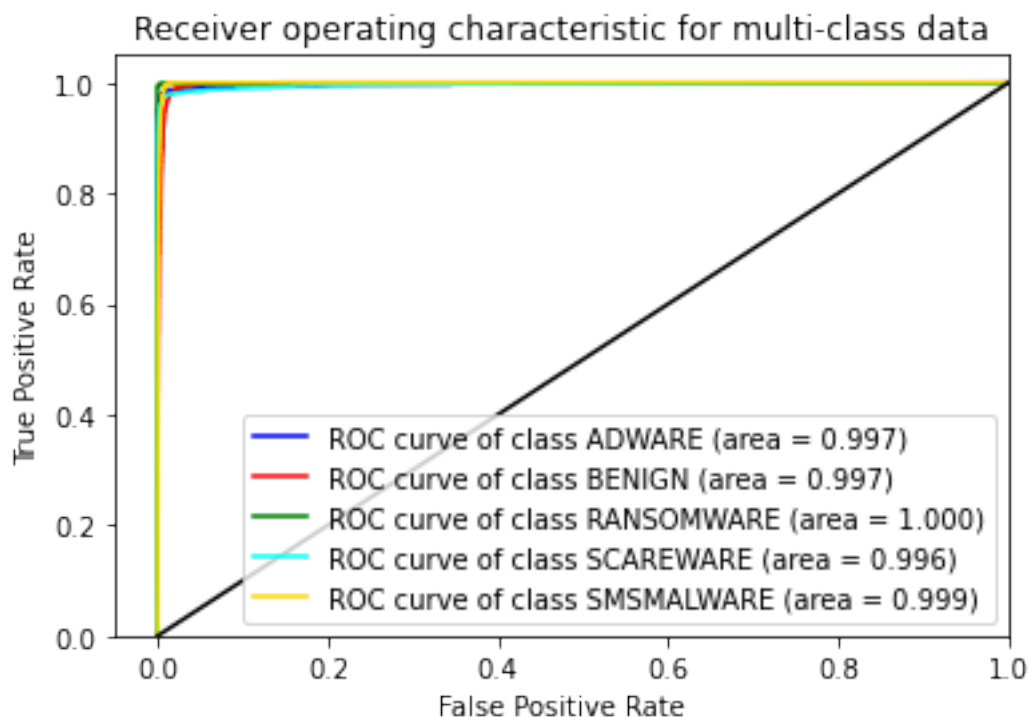
Acurácia: 0.978

Erro: 0.022

```
[118]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_validacao_norm, y_pred,
                                     set(y_validacao_norm),
                                     'Matriz de Confusão RF (PS)')
```



```
[119]: # Exibe curva ROC
funcoes_uteis.plot_roc_curve(modelo_rf_ps, X_train_norm, y_train_norm,
                              X_validacao_norm, y_validacao_norm)
```



4.2.3 MLP (*Multi-layer Perceptron*)

```
[158]: # Obtém o modelo gerado
modelo_mlp_ps = dic_melhores_modelos['split_perc_mlp'][0]
y_pred = modelo_mlp_ps.predict(X_validacao_norm)
print(classification_report(y_pred, y_validacao_norm, zero_division=True))
```

	precision	recall	f1-score	support
ADWARE	0.83	0.78	0.80	2725
BENIGN	0.90	0.81	0.85	8094
RANSOMWARE	0.84	0.91	0.87	1864
SCAREWARE	0.71	0.85	0.77	1924
SMSMALWARE	0.61	0.92	0.74	902
accuracy			0.83	15509
macro avg	0.78	0.85	0.81	15509
weighted avg	0.84	0.83	0.83	15509

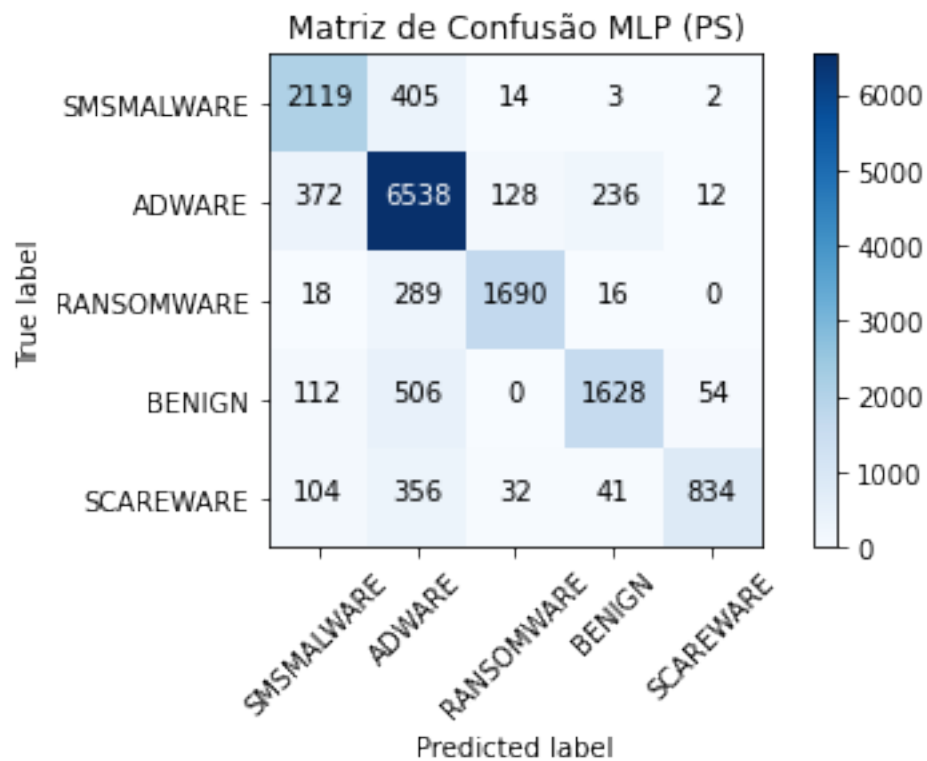
```
[159]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_validacao_norm, y_pred)
erro = 1 - accuracy
```

```
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

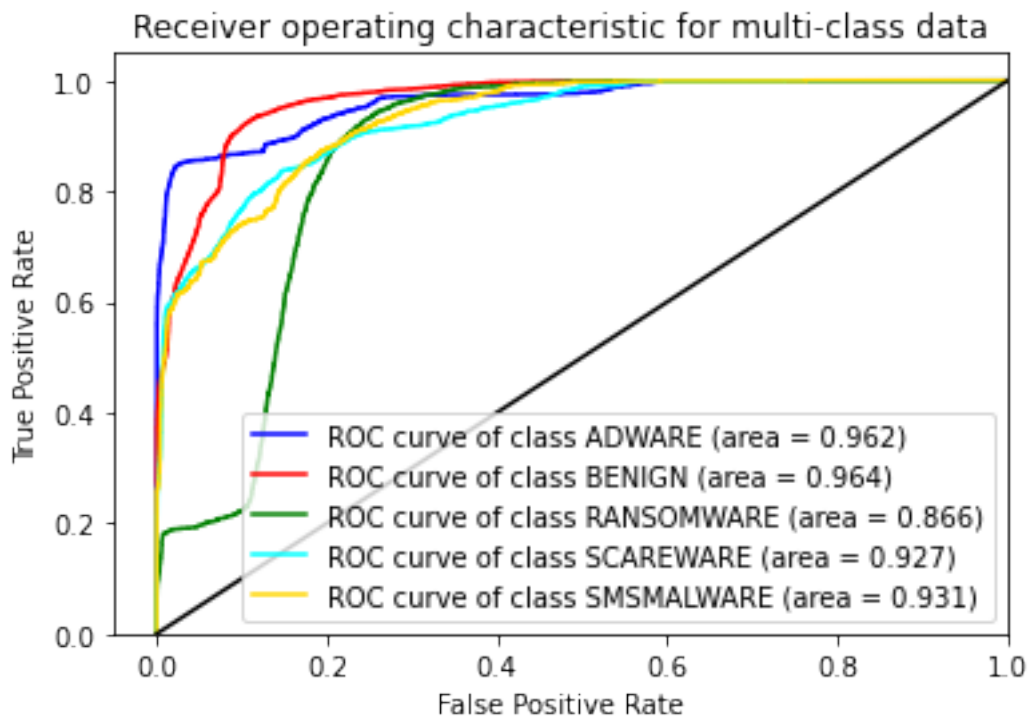
Acurácia: 0.826

Erro: 0.174

```
[160]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_validacao_norm, y_pred,
                                     set(y_validacao_norm),
                                     'Matriz de Confusão MLP (PS)')
```



```
[161]: # Exibe curva ROC
funcoes_uteis.plot_roc_curve(modelo_mlp_ps, X_train_norm, y_train_norm,
                              X_validacao_norm, y_validacao_norm)
```



4.3 Melhores modelos gerados com validação cruzada

4.3.1 KNN (*K-nearest neighbors algorithm*)

```
[45]: # Obtém o melhor modelo da validação cruzada
melhor_modelo_knn_vc = dic_melhores_modelos['melhor_val_cruz_knn'][0]
y_pred = melhor_modelo_knn_vc.predict(X_validacao_norm)
print(classification_report(y_pred, y_validacao_norm, zero_division=True))
```

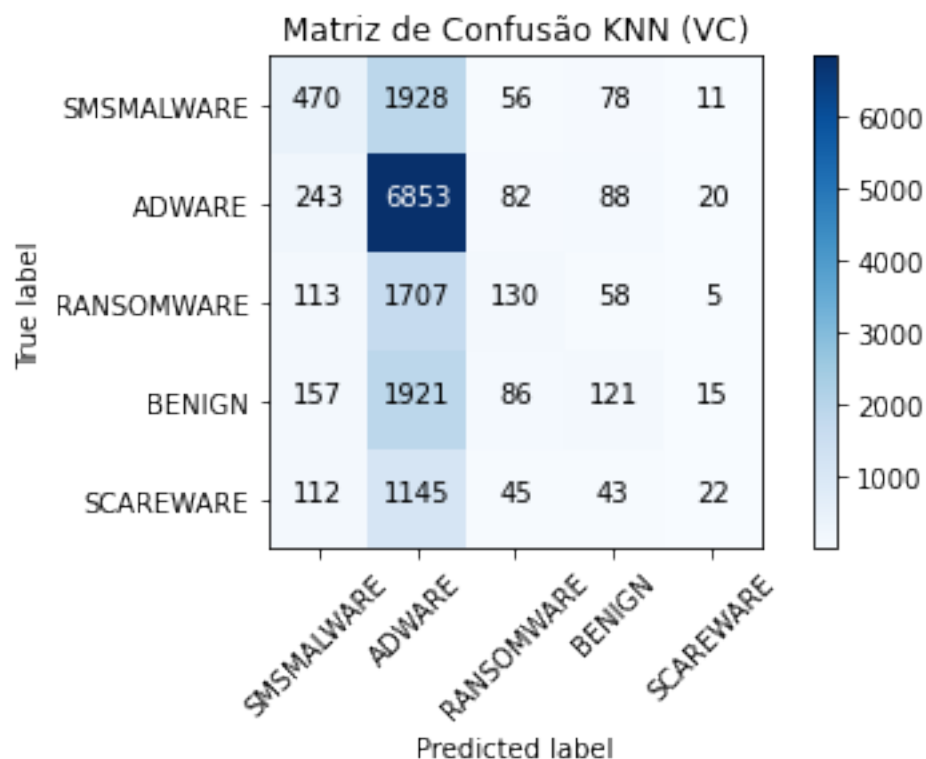
	precision	recall	f1-score	support
ADWARE	0.18	0.43	0.26	1095
BENIGN	0.94	0.51	0.66	13554
RANSOMWARE	0.06	0.33	0.11	399
SCAREWARE	0.05	0.31	0.09	388
SMSMALWARE	0.02	0.30	0.03	73
accuracy			0.49	15509
macro avg	0.25	0.37	0.23	15509
weighted avg	0.84	0.49	0.60	15509


```
[46]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_validacao_norm, y_pred)
erro = 1 - accuracy
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

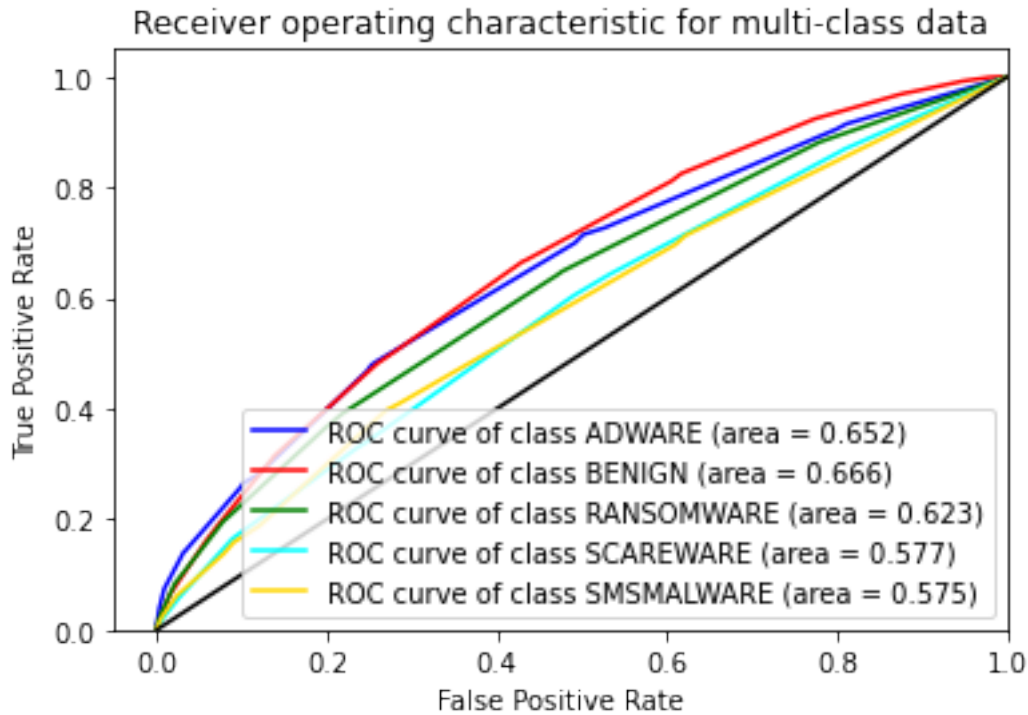
Acurácia: 0.490

Erro: 0.510

```
[47]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_validacao_norm, y_pred,
                                     set(y_validacao_norm),
                                     'Matriz de Confusão KNN (VC)')
```



```
[57]: # Exibe curva ROC
X_train_data = dic_melhores_modelos['melhor_val_cruz_knn'][2][0][0]
y_train_data = dic_melhores_modelos['melhor_val_cruz_knn'][2][0][1]
funcoes_uteis.plot_roc_curve(melhor_modelo_knn_vc, X_train_data, y_train_data,
                              X_validacao_norm, y_validacao_norm)
```



4.3.2 Random Forest

```
[82]: X_train_data = dic_melhores_modelos['melhor_val_cruz_rf'][2][0][0]
      y_train_data = dic_melhores_modelos['melhor_val_cruz_rf'][2][0][1]

      clf_rf = RandomForestClassifier(n_estimators = 50, random_state=0)
      clf_rf.fit(X_train_data, y_train_data)
      melhor_modelo_rf_vc = clf_rf

[83]: # Obtém o melhor modelo da validação cruzada
      #melhor_modelo_rf_vc = dic_melhores_modelos['melhor_val_cruz_rf'][0][0]
      y_pred = melhor_modelo_rf_vc.predict(X_validacao_norm)
      print(classification_report(y_pred, y_validacao_norm, zero_division=True))
```

	precision	recall	f1-score	support
ADWARE	0.97	0.99	0.98	2503
BENIGN	0.99	0.97	0.98	7468
RANSOMWARE	0.99	1.00	0.99	2005
SCAREWARE	0.95	0.99	0.97	2207
SMSMALWARE	0.94	0.97	0.96	1326
accuracy			0.98	15509
macro avg	0.97	0.98	0.98	15509

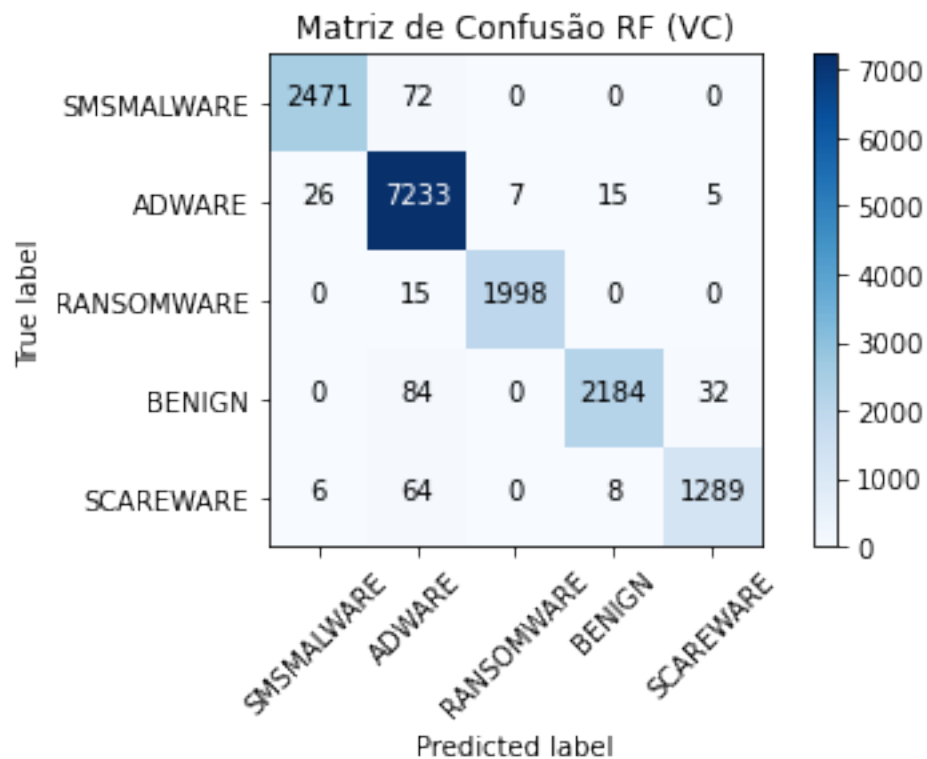
weighted avg 0.98 0.98 0.98 15509

```
[84]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_validacao_norm, y_pred)
erro = 1 - accuracy
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

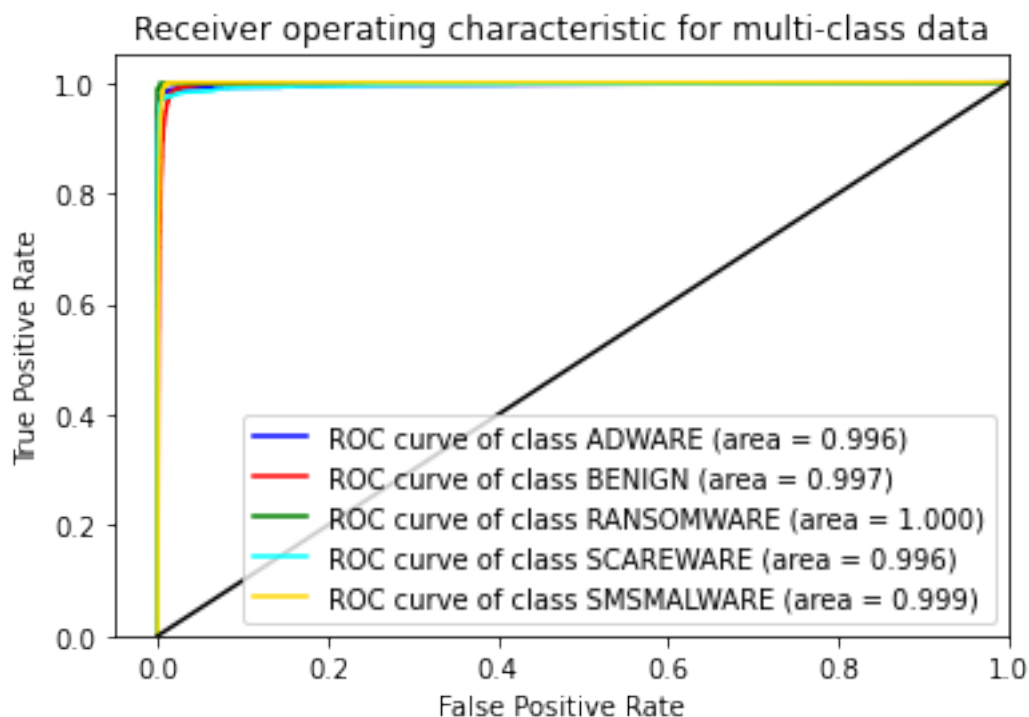
Acurácia: 0.978

Erro: 0.022

```
[85]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_validacao_norm, y_pred,
                                     set(y_validacao_norm),
                                     'Matriz de Confusão RF (VC)')
```



```
[99]: # Exibe curva ROC
X_train_data = dic_melhores_modelos['melhor_val_cruz_rf'][2][0][0]
y_train_data = dic_melhores_modelos['melhor_val_cruz_rf'][2][0][1]
funcoes_uteis.plot_roc_curve(melhor_modelo_rf_vc, X_train_data, y_train_data,
                              X_validacao_norm, y_validacao_norm)
```



4.3.3 MLP (*Multi-layer Perceptron*)

```
[68]: # Obtém o melhor modelo da validação cruzada
melhor_modelo_mlp_vc = dic_melhores_modelos['melhor_val_cruz_mlp'][0][0]
y_pred = melhor_modelo_mlp_vc.predict(X_validacao_norm)
print(classification_report(y_pred, y_validacao_norm, zero_division=True))
```

	precision	recall	f1-score	support
ADWARE	0.88	0.73	0.80	3070
BENIGN	0.93	0.78	0.85	8668
RANSOMWARE	0.82	0.74	0.78	2241
SCAREWARE	0.41	0.92	0.57	1032
SMSMALWARE	0.32	0.88	0.47	498
accuracy			0.78	15509
macro avg	0.67	0.81	0.69	15509
weighted avg	0.85	0.78	0.80	15509

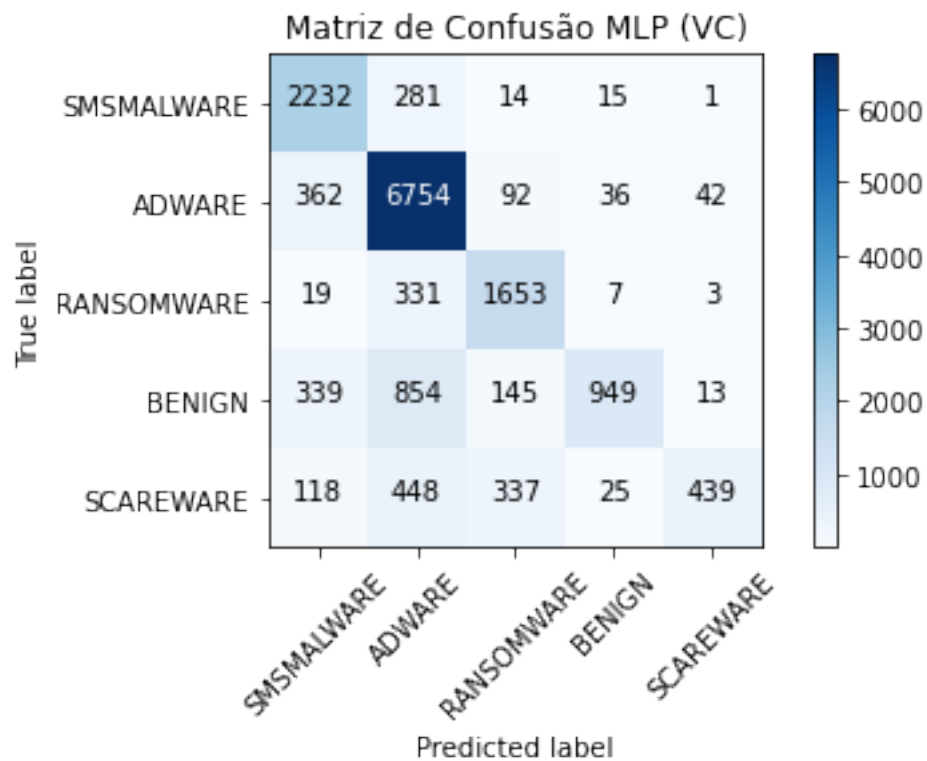
```
[69]: # Calcula erro (1-accuracy)
accuracy = accuracy_score(y_validacao_norm, y_pred)
erro = 1 - accuracy
```

```
print('Acurácia: {:.3f}'.format(accuracy))
print('Erro: {:.3f}'.format(erro))
```

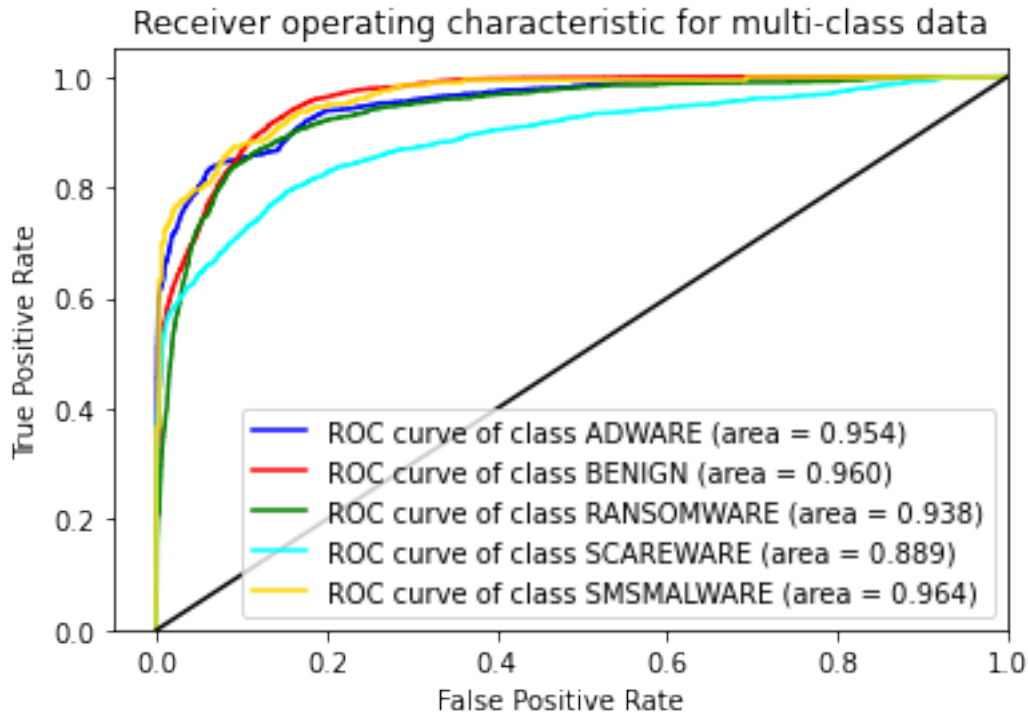
Acurácia: 0.775

Erro: 0.225

```
[70]: # Exibe a matriz de confusão
funcoes_uteis.plot_confusion_matrix(y_validacao_norm, y_pred,
                                     set(y_validacao_norm),
                                     'Matriz de Confusão MLP (VC)')
```



```
[71]: # Exibe curva ROC
X_train_data = dic_melhores_modelos['melhor_val_cruz_mlp'][2][0][0]
y_train_data = dic_melhores_modelos['melhor_val_cruz_mlp'][2][0][1]
funcoes_uteis.plot_roc_curve(melhor_modelo_mlp_vc, X_train_data, y_train_data,
                             X_validacao_norm, y_validacao_norm)
```



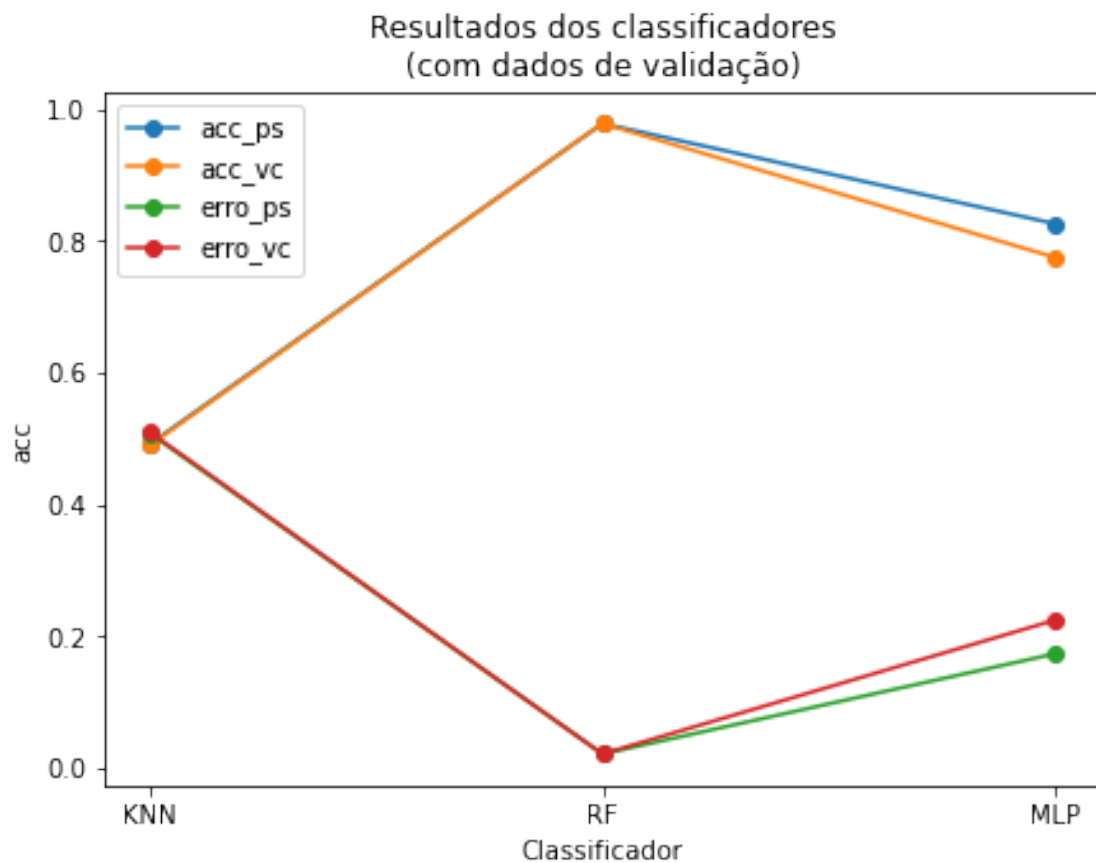
4.3.4 Comparação de Resultados e discussão

- Os resultados obtidos com dados de validação permite concluir que os modelos gerados são consistentes com o erro e acurácia anteriormente obtidos, dado a proximidade dos resultados.

```
[171]: classificador_valid = ['KNN', 'RF', 'MLP', 'KNN', 'RF', 'MLP', 'KNN', 'RF', 'MLP', 'KNN', 'RF', 'MLP']
        acuracias_valid = [0.492, 0.978, 0.826, 1-0.492, 1-0.978, 1-0.826, 0.490, 0.978, 0.775, 1-0.490, 1-0.978, 1-0.775]
        divisao_valid = ['acc_ps', 'acc_ps', 'acc_ps', 'erro_ps', 'erro_ps', 'erro_ps', 'acc_vc', 'acc_vc', 'acc_vc', 'erro_vc', 'erro_vc', 'erro_vc']

        # Exibe os resultados
        funcoes_uteis.plot_resultados(classificador_valid, 'Classificador',
                                      acuracias_valid, 'acc', divisao_valid,
                                      'Resultados dos classificadores\n(com dados de validação)', figsize=(7,5))

        # acc_ps - Acurácia Percentage Split
        # acc_vc - Acurácia Validação Cruzada
```



4.3.5 Sítese dos resultados

Treinamento (<i>Percentage Split</i>)						Treinamento (Validação Cruzada, melhor modelo)					
Acc. Dados de Teste			Acc. Dados de Validação			Acc. Média Dados de Teste			Acc. Dados de Validação		
KNN	RF	MLP	KNN	RF	MLP	KNN	RF	MLP	KNN	RF	MLP
0.490	0.981	0.827	0.492	0.978	0.826	0.489	0.978	0.785	0.490	0.978	0.775